UNIVERSIDADE DA CORUÑA

Departamento de Métodos Matemáticos y de Representación
Área de Ingeniería Cartográfica, Geodésica y Fotogrametría

TESIS DOCTORAL

GeoTextura. Una arquitectura software para la visualización en tiempo real de información bidimensional dinámica georreferenciada sobre modelos digitales 3D de terreno basada en una técnica de mapeado de texturas virtuales

Autor: **Francisco Javier Taibo Pena**
Director: **Luis Antonio Hernández Ibáñez**

A Coruña, diciembre de 2009

Este trabajo está dedicado a Paula y a Nuria, a quienes he robado un tiempo precioso que jamás podré devolverles, y a Sandra, que durante su desarrollo ha soportado doble carga de trabajo.
Siempre estaré en deuda con vosotras.

Agradecimientos

Este trabajo no habría sido posible sin la colaboración y el apoyo de muchas personas. Sirvan estas líneas para expresar mi agradecimiento a todas ellas.

En primer lugar a mis padres, por enseñarme lo más importante de todo, y sin lo cual lo demás no sirve para nada. En cierto modo son los verdaderos autores de esta tesis.

A mi hermano, Miguel, por su cariño y su lealtad inquebrantables, sin duda inmerecidas.

A todo el equipo del VideaLAB, y especialmente a todos los que han trabajado en el proyecto SANTI a lo largo de más de una década. A los que menciono a continuación y a los que sin duda injustamente me estoy dejando en el tintero.

Luis Hernández Ibáñez, investigador principal del grupo y director de esta tesis, que hace más de una década me brindó la posibilidad de trabajar en el campo de los gráficos por computador dentro del grupo VideaLAB.

Manuel Meijide Ferreiro, por los primeros momentos en el laboratorio, compañero en esas jornadas interminables, día y noche, donde el SANTI vio la luz y viajó por el mundo.

Mis agradecimientos más especiales al equipo de informáticos del laboratorio, porque de todos ellos he aprendido muchísimo.

Antonio Seoane Nolasco, compañero de viajes, gruñón del laboratorio, maestro del *cutter* y experto en visualización de terreno (entre otras cosas), autor del sistema predecesor a las geotexturas y la primera persona que comenzó a experimentar con texturas 3D para hacer *clipmaps*.

Alberto Jaspe Villanueva, ex-benjamín del laboratorio, profesional todoterreno y mente inquieta que mantiene una relación amor-odio con la luz.

Rubén López Gómez, sin duda de todos los que se han ido, la persona a quien más echo de menos, en lo profesional y en lo personal. Gracias por todos esos estupendos años, por todo lo que aprendí de ti y sobre todo por tu excelente sentido del humor. Ha sido un honor y un placer trabajar a tu lado.

José Antonio Iglesias Guitián, ex-compañero con un humor "explosivo", que nunca dejará de sorprenderme. Otra persona cuya ausencia todavía se nota en el laboratorio.

Gustavo Fariña y Pablo Montero, los nuevos fichajes del laboratorio, por traer un soplo de aire fresco, su entusiasmo y su dedicación. Es difícil encontrar a gente como vosotros.

A Rocío López Mihura, por ser una excelente compañera y sin embargo amiga, tanto en el laboratorio como en la facultad, siempre dispuesta a ayudar con una sonrisa en la cara, aún cuando la mayoría de las veces no me lo merezca. Y por la compañía y el suministro de cafeína en el mes agosto, durante la recta final de este trabajo.

A Juan José Nieto Boga, por su inagotable paciencia, trabajando rodeado de "informáticos locos", y siempre dispuesto a echar una mano cuando se lo pedimos para "hacer que esto luzca".

A Sandra Martínez Costa, compañera de fatigas el verano anterior, cuando la facultad estaba desierta. Por ser "la guardiana de la puerta" y dejarse sobornar con chocolate.

A Juan Blanco y David Míguez, que animan, mueven y lo que haga falta, pero sobre todo animan.

A mis viejos compañeros del laboratorio RNASA: Jano, Nino, Ana, Julián, Nieves, JJ, Delia, Ana Belén, Juanra, Dafonte, Javi Pereira, Rega y muchos otros que sin duda injustamente me estoy olvidando. Junto a ellos di mis primeros pasos en el mundo de la investigación en la compañía más grata. Gracias por aquellos momentos inolvidables.

A Miguel Barreiro, un gran amigo, compañero de aventuras y excelente profesional de quien he aprendido muchísimo. Gracias por los buenos momentos, las conversaciones interminables y esos locos proyectos diseñados en servilletas de papel.

A todos los compañeros de la Facultad de Ciencias de la Comunicación: Antonio Sanjuán, José Videla, Victoria De León, Teresa Nozal, Olga Osorio, Carmen Costa, Teresa Piñeiro, Ana González y todos los demás. Gracias por hacer de la facultad un lugar tan agradable en el que trabajar.

A todos aquellos alumnos y alumnas que hacen de la docencia una labor atractiva y reconfortante. A Érika González Eguía por sus "monstruitos", su creatividad y su imaginación, a Elena Gómez Quintela por su fe ciega en mi trabajo, su asesoramiento estadístico y sobre todo por traer bombones, a Irene Somoza Sampaio por su simpatía y buen humor.

A Nacho Miranda, por los buenos momentos, cuando SGI aún era Silicon Graphics y el SANTI viajaba por el mundo en una *InfiniteReality*.

A César, de la exposición "Camiño de Santiago Virtual", sin duda el mejor piloto que ha manejado el SANTI.

Por último, a todos los que me he olvidado, y a usted lector, por dedicar su valioso tiempo a leer este trabajo. Espero que haya merecido la pena.

Índice general

1. Introducción **1**
- 1.1. Motivación . . . 1
- 1.2. La importancia de la visualización en los SIG . . . 5
- 1.3. Trabajos previos . . . 7
- 1.4. Aportación de la investigación . . . 14
- 1.5. Estructura de la memoria . . . 16

2. Antecedentes **17**
- 2.1. Introducción histórica . . . 17
- 2.2. Sistemas de información geográfica (SIG) . . . 19
  - 2.2.1. La información geográfica . . . 21
  - 2.2.2. Sistemas de coordenadas . . . 25
  - 2.2.3. Proyecciones cartográficas . . . 29
  - 2.2.4. *Software* SIG . . . 37
- 2.3. Visualización de terreno . . . 40
- 2.4. Modelos topográficos . . . 43
  - 2.4.1. Mallas regulares (*grid*) . . . 44
  - 2.4.2. Mallas irregulares (TIN) . . . 45
- 2.5. Visualización en tiempo real de terreno . . . 48
  - 2.5.1. Paginación y cachés . . . 51
  - 2.5.2. Niveles de detalle . . . 52
  - 2.5.3. Técnicas existentes para la visualización de terreno . . . 54
- 2.6. Sistemas de texturizado de terreno . . . 58
  - 2.6.1. Mapeado de texturas . . . 58
  - 2.6.2. La textura de terreno global . . . 73
  - 2.6.3. *Clipmaps* . . . 79
  - 2.6.4. MP-Grid . . . 84
  - 2.6.5. Otros sistemas de texturizado . . . 85
- 2.7. Datos vectoriales 2D sobre el terreno . . . 97
  - 2.7.1. Construcción de geometría 3D . . . 97
  - 2.7.2. Proyección mediante texturizado . . . 103
  - 2.7.3. Otras aproximaciones . . . 106
- 2.8. Limitaciones de los sistemas actuales . . . 108

2.8.1. VGIS (*Virtual Geographic Information System*) 108
2.8.2. Otros sistemas comerciales 109
2.8.3. Resumen de las técnicas estudiadas 124

3. Hipótesis **135**
3.1. Punto de partida 135
3.2. Objetivos 140
3.2.1. Volumen de información 140
3.2.2. Rendimiento 140
3.2.3. Calidad de la visualización 142
3.2.4. Prestaciones 143
3.2.5. Requisitos de *hardware* 145
3.3. Metodología 146

4. Desarrollo **149**
4.1. Arquitectura general 150
4.1.1. Núcleo del sistema de texturizado dinámico 151
4.1.2. Capa de adaptación 152
4.1.3. Capa de acceso a datos 153
4.1.4. Entorno de desarrollo 155
4.2. Interfaz del sistema (MotorTextura) 156
4.3. Textura virtual dinámica (GeoTextura) 158
4.3.1. Coordenadas de textura 160
4.3.2. CacheTRAM 162
4.3.3. Estructura de la caché 164
4.3.4. *Render* 167
4.4. Actualización síncrona 168
4.4.1. Orden de carga de los niveles 171
4.4.2. Orden de carga de las teselas dentro de un nivel 172
4.4.3. Gestión de los metadatos de la caché en TRAM 175
4.4.4. Actualización temporal 177
4.4.5. Continuidad en los bordes de la textura virtual 178
4.4.6. Proceso de actualización de una tesela 179
4.4.7. Tamaño de bloque para la actualización 180
4.5. Actualización asíncrona 182
4.5.1. Arquitectura de la actualización asíncrona 188
4.6. Gestión de datos vectoriales 193
4.6.1. Arquitectura de las cachés para la gestión de datos vectoriales 193
4.6.2. Optimización de los datos vectoriales 195
4.7. Escalabilidad 207

5. Resultados **211**
5.1. *Render* . . . 213
5.1.1. Elección de los entornos de prueba . . . 213
5.1.2. Comparación entre OpenGL y GeoTextura . . . 214
5.1.3. Impacto de la ventana cacheada en el rendimiento del *render* . . . 218
5.1.4. Comparación de calidad entre OpenGL y GeoTextura 220
5.1.5. Conclusiones . . . 221
5.2. Actualización síncrona . . . 221
5.2.1. Recarga de la caché completa . . . 222
5.2.2. Actualización de teselas . . . 224
5.2.3. Impacto del formato de pixel en el rendimiento de la actualización síncrona . . . 228
5.2.4. Conclusiones . . . 231
5.3. Actualización asíncrona . . . 232
5.3.1. Descripción de la caché de referencia . . . 232
5.3.2. Prueba 1. Disco local, sin compresión. . . . 233
5.3.3. Prueba 2. Compresión JPEG . . . 242
5.4. Tratamiento de datos vectoriales . . . 249
5.4.1. Descripción de la batería de pruebas . . . 250
5.4.2. Resultados de las pruebas . . . 253
5.4.3. Conclusiones . . . 261
5.5. Ejemplos de aplicación . . . 262
5.5.1. Demostrador SIG 3D genérico . . . 262
5.5.2. Sistema integral de gestión de tráfico . . . 267
5.5.3. Monitorización de infraestructuras . . . 270
5.5.4. Presentación pública de actuaciones locales . . . 271
5.5.5. Análisis y presentación de cobertura radioeléctrica . . 272
5.5.6. Plan de ordenación del litoral . . . 275

6. Conclusiones **279**
6.1. Futuras líneas de investigación . . . 280
6.1.1. Optimización de la actualización . . . 281
6.1.2. Autoconfiguración . . . 281
6.1.3. Manejo de información vectorial . . . 282
6.1.4. Mejoras en calidad visual . . . 283

A. Interfaces del motor de texturizado **285**
A.1. Interfaz genérica. MotorTextura . . . 285
A.2. Interfaz de textura abstracta. TexturaUnica . . . 287
A.3. Combinación de texturas y *shaders*. MultiTextura . . . 287
A.4. Interfaz de la clase GeoTextura . . . 289
A.5. Interfaz de la clase CacheTRAM . . . 290

B. El proceso de *render* de una GeoTextura **293**
B.1. Filtrado isotrópico 295
B.2. Funciones auxiliares 296
B.3. Filtrado por proximidad 303
B.4. Filtrado bilineal 303
B.5. Filtrado trilineal 305
B.6. Filtrado anisotrópico 306
B.7. Parámetros utilizados por el *shader* 309
B.8. Comparación entre OpenGL y GeoTextura 311

C. Configuración dependiente de la vista **315**
C.1. Ubicación del centro de detalle 315
C.1.1. Simulación de vuelo 315
C.1.2. Sistemas de realidad virtual 318
C.2. Ubicación de la pirámide flotante 319
C.3. Cálculo del nivel de detalle principal 321

D. Escalabilidad del sistema **323**
D.1. Modificaciones a la arquitectura y la interfaz 323
D.2. Modificaciones al proceso de *render* 324
D.3. Modificaciones a la actualización 327

E. CargadorTRAM **331**
E.1. Datos *raster*. CargadorTRAM3DTexSubImage 332
E.2. Datos vectoriales. CargadorTRAM3DRender 333

F. CacheRAM **337**
F.1. Cachés de información *raster* 337
F.2. Cachés de información vectorial 339

G. Acceso a las fuentes de datos **341**
G.1. CargadorRAMClipmapSGI 341
G.2. CargadorRAMGDAL 343
G.3. CargadorRAMWMS 343
G.4. CargadorRAMSceneGraphOSG 345
G.5. CargadorRAMSceneGraphOSGGIS 345
G.6. CargadorRAMSceneGraphGisinho 346
G.7. Resumen de los cargadores disponibles 346

H. Gestión de la carga del sistema **347**
H.1. La actualización dentro del ciclo de *render* 347
H.2. Subsistema de control de carga del *render* 349

I. Mediciones de rendimiento **351**
I.1. Rendimiento del *render* 351
I.1.1. Tiempos de *render* por test 351
I.1.2. Tiempos y fps por configuración 355

J. Comparativa de calidad de imagen **359**

Índice de figuras

1.1. Arquitectura general del sistema 3
1.2. Uso combinado del sistema de visualización con un SIG utilizando múltiples pantallas 4
1.3. Esquema general del sistema de visualización y el acceso a los diferentes tipos de datos 5
1.4. Vistas 3D de los entornos urbanos a nivel de suelo (1999) . . 9
1.5. Imagina 2000 (Montecarlo). Instalación del SANTI en el Reality Center de Silicon Graphics Inc. 10
1.6. Vistas del trazado de la línea de ferrocarril de alta velocidad y representación de las zonas naturales protegidas (2000) . . . 11
1.7. Comparativa de calidad entre bases de datos procedentes de imágenes de satélite y de ortofotografías aéreas 13

2.1. Mapa realizado por John Snow para ubicar los casos del brote de cólera surgido en el Soho londinense en 1854 20
2.2. Ejemplos de información SIG continua en formato *raster*. a) Fotografía aérea. b) Imagen de satélite (espectro visible). c) Imagen de satélite multiespectral (LANDSAT 7). d) Imagen de sensor infrarrojo. e) Imagen de radar. f) Mapa temático de temperaturas. g) Mapa de profundidad de nieve en la Antártida h) Mapa de temperatura atmosférica (monocromático) i) Mapa de temperatura atmosférica (pseudocolor) (Fuentes de las imágenes: SIGPAC, NASA/GSFC/METI/ERSDAC/JAROS, y U.S./Japan ASTER Science Team) 23
2.3. Ejemplos de información vectorial manejada por un SIG. a) Red de carreteras. b) Poblaciones de interés y ruta turística c) Área de un municipio d) Trazado del enlace de una vía rápida. e) y f) Información catastral. 24
2.4. Elipsoide y sistema de coordenadas geográficas (λ,ϕ,h) y cartesianas (X,Y,Z). 27
2.5. Representación del geoide, el código de colores muestra las variaciones en la gravedad (fuente: NASA) 28

2.6. Mapa de la geografía conocida en la época del Imperio Romano, realizado por Ptolomeo en el año 150 (reconstruido en el siglo XV). 30
2.7. Tipos de proyección: cilíndrica, cónica y plana o acimutal (fuente: Escuela Universitaria de Ingeniería Técnica Topográfica de la Universidad Politécnica de Madrid). 31
2.8. Tipos de proyección acimutal o plana (fuente: Escuela Universitaria de Ingeniería Técnica Topográfica de la Universidad Politécnica de Madrid). 31
2.9. Proyección UTM. 32
2.10. División de zonas de la proyección UTM. 33
2.11. Proyección Gall-Peters. 34
2.12. Proyecciones cartográficas. De izquierda a derecha y arriba a abajo: Mercator, Mercator transversa, Gall-Peters, Plate Carée, Cassini, Conforme de Lambert, Cónica equidistante y Albers (fuente: USGS [69]). 35
2.13. Proyecciones cartográficas. De izquierda a derecha y arriba a abajo: Bonne, Estereográfica, Equivalente de Lambert, Postel, Gnomónica y Ortográfica (fuente: USGS [69]). 36
2.14. Simuladores de vuelo. Link Trainer (izquierda) es uno de los sistemas pioneros, anterior a la simulación visual utilizando computadores digitales. A la derecha un simulador actual con movimiento de alta precisión y 6 grados de libertad (©2005 FlightSafety International). 40
2.15. Ejemplos de mallas regular e irregular (TIN). 44
2.16. Problemas de las mallas regulares en zonas con cambios bruscos de pendiente. 46
2.17. Triangulación de Delaunay (negro) y celdas de Voronoi (rojo). 47
2.18. Uso de TIN para adaptar el modelo digital del terreno a las infraestructuras modeladas en 3D. (fuente: VideaLAB) 49
2.19. Conversión de espacios de coordenadas en el mapeado de texturas. 59
2.20. Proceso de mapeado de texturas. 61
2.21. Filtrado bilineal. 63
2.22. Comparativa de técnicas de filtrado para el aumento de la textura. a) Textura aumentada con filtrado por proximidad (*nearest*) b) Textura aumentada con filtrado bilineal. 64
2.23. Efecto *moiré* producido por un remuestreo reduciendo la imagen original tomando muestras por el método de proximidad. A la izquierda la imagen original y a la derecha la versión reducida. 65
2.24. Estructura piramidal con los diferentes niveles de detalle prefiltrados de una textura (*mipmap*). 66

2.25. Comparativa de técnicas de filtrado para la reducción de la textura. a) Filtrado por proximidad b) Fitrado bilineal c) Filtrado trilineal (*mipmap*). 67
2.26. Efecto borroso al aplicar la técnica de *mipmap* en una situación de anisotropía (izquierda) y versión corregida mediante filtrado anisotrópico (derecha). 68
2.27. *Ripmap*. 69
2.28. Esquema de funcionamiento de la técnica de tabla de suma de área. 70
2.29. Gestión del área de un pixel en espacio textura por parte de las técnicas de filtrado anisotrópico. 71
2.30. Organización de la memoria en la textura de terreno global de Cosman. 77
2.31. Estructura de un *clipmap*. 80
2.32. Direccionamiento toroidal. 82
2.33. Problema de la organización en *quadtrees* de los mosaicos de texturas. 89
2.34. Ejemplos de representación de información SIG puntual. . . . 98
2.35. Problemas en la adaptación de líneas 2D sobre el terreno 3D. 98
2.36. Adaptación de líneas 2D sobre el terreno 3D. 99
2.37. Problemas en la visualización de polígonos coplanares. a) Efecto *Z-fighting* debido a errores de precisión en el *Z-buffer* b) Visualización correcta 99
2.38. Filtrado anisotrópico (Google Earth Pro 4.2). 111
2.39. Creación de polígono plano sobre el terreno (Google Earth Pro 4.2). 112
2.40. Visualización de la información SIG vectorial mediante textura (Microsoft Virtual Earth 3D). 113
2.41. Problemas de *aliasing* en el texturizado (Microsoft Virtual Earth 3D). 114
2.42. Problemas de visualización de la información SIG vectorial como geometría, (arriba) vectores atravesando el terreno, (abajo) vectores flotando sobre el terreno. 115
2.43. Visualización de información dinámica (NASA World Wind). 116
2.44. Visualización de información vectorial *rasterizada* sin ordenar en profundidad (NASA World Wind). 117
2.45. Visualización de información vectorial mediante geometría (Geovirtual GeoShow3D). 119
2.46. Problemas de *aliasing* por el filtrado bilineal de las texturas (ArcGIS Explorer). 121
2.47. Creación interactiva de polígonos sobre el terreno (ArcGIS Explorer). Arriba: edición interactiva mediante geometría. Abajo: Resultado final mediante textura. 122
2.48. Carga desordenada de niveles del mosaico de texturas. 125

2.49. Carga desordenada de niveles del mosaico de texturas (NASA World Wind). 126
2.50. Rotura de la coherencia espacial en las uniones de texturas. . 127
2.51. Rotura de la coherencia espacial en las uniones de texturas (continuación). 128
2.52. Rotura de la coherencia espacial debido al montaje de imágenes de distintas fuentes sin ecualizar. 129
2.53. Problemas en la geometría del terreno. 130

3.1. Análisis de las características de la información (dinamismo vs tamaño). 138
3.2. Ejemplo 1. Fotografía aérea de Galicia a 0,25 m/pixel mapeada sobre el modelo digital de elevación del terreno ($\sim$ 3 TBytes sin compresión). 139
3.3. Ejemplo 2. Red carreteras de Galicia con información de nivel de tráfico ($\sim$ 15 MBytes). Datos vectoriales aplicados sobre los datos *raster* (fotografía aérea). 139
3.4. Latencia con un *pipeline* de tres fases en paralelo. 142

4.1. Estructura general e interfaces del sistema propuesto. 151
4.2. Esquema genérico de acceso a los datos 154
4.3. Interfaz de texturizado 156
4.4. Arquitectura global del sistema de gestión de texturas virtuales dinámicas (GeoTextura). 159
4.5. Programa por vértice para la autogeneración de las coordenadas de textura. 161
4.6. Programa por vértice para el paso de coordenadas de textura junto con la información de los vértices. 161
4.7. CacheTRAM3D. Diagrama UML de las clases CacheTRAM y relacionadas. 162
4.8. CacheTRAM3D. Distribución de la caché con estructura de *clipmap* en las capas de una textura 3D. 165
4.9. División del espacio virtual en teselas, ventana cacheada, direccionamiento toroidal, y desplazamientos local y global. . . 167
4.10. Orden de carga de niveles. a) Aproximación clásica. b) Aproximación propuesta para datos dinámicos. 172
4.11. Orden de carga de las teselas dentro de un nivel de detalle. En este jemplo se muestra la carga de los tres primeros anillos concéntricos de teselas para un nivel determinado asumiendo una posible dirección de avance. 174
4.12. División de un anillo concéntrico en cuatro brazos y orden de carga de sus teselas para una dirección de avance de ejemplo. 174
4.13. Diagrama UML de las clases que encapsulan el estado de la caché. 176

4.14. Ejemplos de ubicación de la ventana cacheada según las condiciones de continuidad en los bordes. 179
4.15. Tiempos de carga de bloques en TRAM y velocidades de transferencia. NVIDIA GeForce 7800 GS/AGP/SSE2 (OpenGL 2.1.0 NVIDIA 97.46 Linux x86). 182
4.16. Tiempos de carga de bloques en TRAM y velocidades de transferencia. NVIDIA GeForce 8800 GTS/PCI/SSE2 (OpenGL 2.1.0 NVIDIA 97.55 Linux x86). 183
4.17. Comportamiento del algoritmo de asignación de prioridad de carga a los *buffers* de `CacheRAM` para un ejemplo de dirección de avance del centro de detalle. 185
4.18. Estrategia 1 para la carga de zonas entre niveles. Disponibilidad rápida. 187
4.19. Estrategia 2 para la carga de zonas entre niveles. Máxima estabilidad. 187
4.20. Arquitectura del sistema de actualización asíncrona de la caché predictiva. 188
4.21. Proceso de una petición de *buffer* por un *thread* de precarga. 191
4.22. Problemas en el grosor de línea debido al límite de OpenGL en cada nivel de detalle de la textura. 201
4.23. Líneas sueltas convertidas a polígonos sin tener en cuenta su continuidad o la conexión entre ellas. 203
4.24. Líneas sueltas convertidas a polígonos teniendo en cuenta la continuidad y las conexiones entre ellas. 203
4.25. Capa vectorial proyectada sobre textura. Filtrado por proximidad. 205
4.26. Capa vectorial proyectada sobre textura. Filtrado trilineal. . . 205
4.27. Capa vectorial proyectada sobre textura. Filtrado anisotrópico con 64 muestras. 206
4.28. Capa vectorial proyectada sobre textura. Filtrado anisotrópico con 64 muestras y suavizado de líneas (*alpha blending + line smooth*). 206
4.29. Aspecto visual de los niveles de mayor detalle de la textura cuando se supera el límite de precisión manejado por el *hardware* (32 bits). 208
4.30. Estructura de pila virtual. Se cachea una ventana de niveles de la pila (pila flotante). 209
4.31. Organización en memoria física de los niveles cacheados de la pila. 210

5.1. Herramienta para demostración y análisis del motor de texturizado virtual dinámico: Earthfly. 212
5.2. Tiempos de *render* por test en las diferentes configuraciones (ms, escala logarítmica). 217

5.3. Comparación de tiempos de *render* con OpenGL y GeoTextura (Bruinen, Caradhras, Shelob y Shelob con SLI). 218
5.4. Tiempo de *render* por muestra anisotrópica (Bruinen, Caradhras, Shelob y Shelob con SLI). 219
5.5. Tiempo de *render* por tamaño de ventana cacheada (Bruinen, Caradhras y Shelob). 220
5.6. Tiempo de recarga de la caché completa (Bruinen, Caradhras, Shelob y Shelob con SLI). 223
5.7. Tasa de transferencia de la recarga de la caché completa (Bruinen, Caradhras, Shelob y Shelob con SLI). 224
5.8. Tiempo medio de actualización de una tesela (Bruinen, Caradhras, Shelob y Shelob con SLI). Izquierda: escala lineal, derecha: escala logarítmica. 225
5.9. Tasa de actualización de una tesela (Bruinen, Caradhras, Shelob y Shelob con SLI). Izquierda: escala lineal, derecha: escala logarítmica. 227
5.10. Tasa de actualización de una tesela según formato de pixel (Bruinen). Tamaños de ventana: 512, 1024 y 2048 *texels*. . . . 229
5.11. Tasa de actualización de una tesela según formato de pixel (Caradhras). Tamaño de ventana: 512, 1024 y 2048 *texels*. . . 230
5.12. Tasa de actualización de una tesela según formato de pixel (Shelob). Tamaño de ventana: 512, 1024 y 2048 *texels*. 230
5.13. Ratio de aciertos de caché (Bruinen, Caradhras y Shelob). Izquierda: por tamaño de *buffer*, derecha: por velocidad. . . . 236
5.14. Nivel de actualización de la caché en TRAM (Briunen, Caradhras y Shelob). Izquierda: por tamaño de *buffer*, derecha: por velocidad. 239
5.15. Ratio de actividad de los *threads* de cargas asíncronas, por tamaños de *buffer* y por velocidades de vuelo. Bruinen, Caradhras y Shelob. 241
5.16. Ratio de aciertos según velocidad. Bruinen, Caradhras y Shelob.245
5.17. Nivel de actualización de la caché en TRAM según velocidad. Bruinen, Caradhras y Shelob. 246
5.18. Desbordamientos del tiempo de cuadro según velocidad. Bruinen, Caradhras y Shelob. 247
5.19. Muestra del conjunto de datos vectoriales utilizados en las pruebas. 250
5.20. Estructura del scenegraph en las configuraciones de los tests 2, 3 y 4. 251
5.21. Tiempo de regeneración de la caché completa (Bruinen). Tamaños de ventana: 512 y 1024. 259
5.22. Tiempo de regeneración de la caché completa (Caradhras). Tamaños de ventana: 512, 1024 y 2048. 259

5.23. Tiempo de regeneración de la caché completa (Shelob). Tamaños de ventana: 512, 1024 y 2048. 260
5.24. Tiempos medios de las diferentes fases de OSG en la generación de una tesela. Tests 1 y 2. 260
5.25. Demostrador genérico de SANTI y GeoTextura. Ortofotografía mezclada con la información vectorial. 263
5.26. Demostrador genérico de SANTI y GeoTextura. Información vectorial sobre la ortofotografía y sobre un color sólido de fondo. 264
5.27. Interfaz de selección y configuración de capas SIG. 266
5.28. Aplicación de gestión y monitorización de información en tiempo real del Centro de Gestión de Tráfico del Noroeste (A Coruña). 268
5.29. Aplicación para la gestión y monitorización de infraestructuras de suministro de agua. 270
5.30. Interfaz para la reproducción de secuencias de texturas virtuales. 271
5.31. Aplicación para la presentación de información geográfica mediante interfaz multitáctil. 272
5.32. Aplicación para el análisis de cobertura de TDT en Galicia. . 274
5.33. Aplicación para la visualización del Plan de Ordenación del Litoral de Galicia. 276

A.1. Programa por vértice por omisión para un objeto MultiTextura. 288
A.2. Programa por *fragment* por omisión para un objeto MultiTextura. 288
A.3. Diagrama UML de la clase GeoTextura. 289

B.1. Proyección de un *texel* del nivel de mayor detalle sobre el espacio pantalla y cálculo de los valores `dx`, `dy`, `hx` y `hy`. . . . 295
B.2. Ajuste de las coordenadas de textura en los extremos sin continuidad o en la vecindad de otros niveles. 298
B.3. Defecto visual por interpolación en los bordes de las ventanas de caché de los niveles de la pila. 302
B.4. Filtrado por proximidad con los niveles marcados con un código de colores. 304
B.5. Filtrado bilineal con los niveles marcados con un código de colores. 305
B.6. Filtrado trilineal con los niveles marcados con un código de colores. 307
B.7. Ejemplo de filtrado anisotrópico mediante tres muestras trilineales. 309
B.8. Filtrado anisotrópico con los niveles marcados con un código de colores. 310
B.9. Comparativa de calidad entre los diferentes filtros implementados: proximidad, bilineal y trilineal. 313

B.10. Comparativa de calidad entre los diferentes filtros implementados: anisotrópico con límites de 4, 8 y 64 muestras. 314

C.1. Ubicación del centro de detalle con la textura alineada con el espacio pantalla. 316
C.2. Ubicación del centro de detalle con la textura no alineada con el espacio pantalla. 317
C.3. Cálculo de la distancia visible en pantalla (en unidades del espacio mundo). 320

D.1. Desplazamiento al nivel de la pirámide direccionado. 327
D.2. Transformación de las coordenadas de los límites de la zona válida cacheada desde el espacio virtual completo al espacio direccionable por el *shader*. 328

E.1. Diagrama UML de la clase `CargadorTRAM`. 331

F.1. Diagrama UML de la clase `CacheRAM`. 337

G.1. Diagrama UML de la clase `CargadorRAM`. 342
G.2. Diagrama UML de la clase `CargadorRAMWMS`. 344

H.1. Diferentes opciones para el orden de las tareas necesarias para la generación de un cuadro de la animación. 348

I.1. Tests 1 a 6. Sin textura, OpenGL: filtrado por proximidad, bilineal, trilineal, anisotrópico 2x y anisotrópico 4x. 352
I.2. Tests 7 a 12. OpenGL: filtrado anisotrópico 8x y 16x, GeoTextura: filtrado por proximidad, bilineal, trilineal y anisotrópico 1x. 353
I.3. Tests 13 a 18. GeoTextura: Filtrado anisotrópico 2x, 4x, 8x, 16x, 32x y 64x. 354

J.1. GeoTextura. Filtrado por proximidad. 360
J.2. OpenGL. Filtrado por proximidad. 360
J.3. GeoTextura. Filtrado bilineal. 361
J.4. OpenGL. Filtrado bilineal. 361
J.5. GeoTextura. Filtrado trilineal. 362
J.6. OpenGL. Filtrado trilineal. 362
J.7. GeoTextura. Filtrado anisotrópico con dos muestras. 363
J.8. OpenGL. Filtrado anisotrópico con dos muestras. 363
J.9. GeoTextura. Filtrado anisotrópico con cuatro muestras. . . . 364
J.10. OpenGL. Filtrado anisotrópico con cuatro muestras. 364
J.11. GeoTextura. Filtrado anisotrópico con ocho muestras. 365
J.12. OpenGL. Filtrado anisotrópico con ocho muestras. 365
J.13. GeoTextura. Filtrado anisotrópico con dieciséis muestras. . . 366

J.14. OpenGL. Filtrado anisotrópico con dieciséis muestras. 366
J.15. GeoTextura. Filtrado anisotrópico con treinta y dos muestras. 367
J.16. GeoTextura. Filtrado anisotrópico con sesenta y cuatro muestras. 367

Índice de cuadros

2.1. Clasificación de la información geográfica según sus características. 24
2.2. Modelos de elipsoide más importantes. 27
2.3. *Data* geodésicos. 28
2.4. Principales proyecciones cartográficas y sus características. . . 31
2.5. Comparativa de uso de memoria de las diferentes técnicas de filtrado de reducción de texturas. 69
2.6. Comparativa de técnicas de texturizado de terreno. 132
2.7. Comparativa de técnicas de visualización de información SIG 2D sobre el terreno 3D. 133

5.1. Configuraciones de *hardware* utilizadas para las pruebas de rendimiento del *render*. 215
5.2. Primera batería de pruebas de rendimiento del *render*. Configuración de cada test. 215
5.3. Comparación de rendimiento (fotogramas por segundo). . . . 216
5.4. Estimación del tiempo de texturizado (ms) con OpenGL y GeoTextura. 216
5.5. Comparación de tiempo de *render* (ms) entre filtrado trilineal y anisotrópico con una sola muestra. 217
5.6. Formatos de pixel utilizados en las pruebas de transferencia. . 228
5.7. Velocidades de vuelo de las pruebas de caché. 234
5.8. Porcentajes de tiempo de actividad de los *threads* de precarga. 240
5.9. Tiempos medios de carga y descompresión. 242
5.10. Porcentaje de aciertos de caché (Bruinen). 244
5.11. Porcentaje de aciertos de caché (Caradhras). 244
5.12. Porcentaje de aciertos de caché (Shelob). 244

5.13. Configuraciones de test de datos vectoriales. D: descomposición de primitivas, QT: árbol cuaternario (*quadtree*), LODs: umbral de visibilidad en pixels/*texels*, DL: uso de *display lists*, VBO: uso de *vertex buffer objects*: Prims=tipo de primitiva para líneas (TS=tristrips, Q=quads, QS=quadstrips), AA: suavizado de puntos/líneas/polígonos, T: tiempo de preproceso en segundos, Tamaño: tamaño del fichero resultante en bytes. 253
5.14. Número de nodos, vértices y primitivas de los tests 7, 9 y 11. 255
5.15. Tiempos (en ms) de generación de tesela en las diferentes pruebas (Bruinen). 256
5.16. Tiempos (en ms) de generación de tesela en las diferentes pruebas (Caradhras). 257
5.17. Tiempos (en ms) de generación de tesela en las diferentes pruebas (Shelob). 258

B.1. Correspondencia entre los filtros de GeoTextura y los de OpenGL.312

G.1. Resumen de cargadores en RAM disponibles en el desarrollo actual. 346

I.1. Rendimiento del *render* (Bruinen). 355
I.2. Rendimiento del *render* (Caradhras). 356
I.3. Rendimiento del *render* (Shelob). 356
I.4. Rendimiento del *render* (Shelob con SLI). 357

Capítulo 1

Introducción

1.1. Motivación

Los sistemas de información geográfica (en adelante SIG) son un campo de estudio que ha experimentado un gran auge en las últimas décadas. Sin embargo, a pesar del grado de madurez alcanzado, actualmente todavía carecen de buenas herramientas para la visualización en 3D de forma interactiva. El tiempo de respuesta, la velocidad de refresco de la pantalla al desplazarse por un área geográfica extensa y la calidad de la visualización no son tan buenos como sería deseable, sobre todo a medida que aumenta el volumen y detalle de los datos visualizados. Además, la vista de los datos tradicionalmente se realiza mediante una vista 2D en planta.

En los últimos años han surgido algunos productos que suponen grandes avances en este aspecto. Se pueden citar ejemplos como Google Earth [79], Microsoft Virtual Earth 3D [53], NASA World Wind [32], TerrainView Globe de ViewTec [29], o TerraExplorer de Skyline [26]. Estos productos permiten navegar sobre una vista tridimensional del terreno con aspecto bastante realista. Sin embargo, dentro de los SIG utilizados habitualmente para el análisis y tratamiento de información geoespacial, todavía no está muy extendido su uso, y los pocos sistemas disponibles de este tipo adolecen de muchas carencias y limitaciones en cuanto a la calidad de visualización y la agilidad de la navegación. Ejemplos de aplicaciones de este tipo que han comenzado a surgir en los últimos años son ArcGIS Explorer o ArcGlobe (incluido en el producto ArcGIS 3D Analyst) de ESRI [7], MetaVR 3D Layering Control Plugin for ArcMap [113], o Leica Virtual Explorer de Leica Geosystems [13].

En el ámbito de la investigación, en contraposición a los productos comerciales anteriormente mencionados, el ejemplo más notorio es VGIS (*Virtual Geographic Information System*) [99, 102], desarrollado en el Georgia Institute of Technology. Este sistema, cuyo desarrollo comenzó a finales de la década de los 90, es uno de los pioneros en la integración de los SIG con la visualización interactiva de terreno en 3D.

La cantidad de herramientas de este tipo que están surgiendo, así como la gran aceptación que están teniendo entre los usuarios a todos los niveles, es una clara prueba de la enorme demanda tecnológica para sistemas que resuelvan los problemas existentes, reduciendo los límites a la agilidad en el manejo de grandes volúmenes de datos y la calidad de la visualización de los mismos.

Por lo tanto, se hace patente la necesidad de disponer de un sistema de visualización de terreno en tiempo real que combine información dinámica en forma de mapas *raster* y datos vectoriales con una calidad realista y un tiempo de respuesta adecuado a las aplicaciones que demanda el mercado. Este sistema debe ser lo suficientemente versátil y abierto como para interoperar con SIGs o cualquier otro tipo de aplicación externa, así como alimentarse de cualquier fuente de datos geoespaciales, ya sea local o remota.

El objetivo del presente trabajo es el desarrollo de un sistema de visualización en tiempo real de terreno que combine información de tipo *raster* (fotografías aéreas, imágenes capturadas desde satélites, etc) con información vectorial procedente de SIGs. Estas informaciones combinadas se aplicarán sobre un modelo digital de terreno compuesto por datos de elevación (ya sea en forma de mallas regulares, TINs o cualquier otra estructura). La información aplicada sobre la geometría del terreno puede ser de carácter dinámico, de forma que deberá ser periódicamente actualizada, ya sea a partir de datos capturados en tiempo real o bien a partir de archivos de datos históricos para visualizar la evolución en el tiempo de la información estudiada.

Este trabajo se centra en el sistema de gestión dinámica de las imágenes *raster* y los datos vectoriales aplicados sobre el terreno, dejando aparte la gestión de la geometría del terreno, que se podrá resolver con cualquiera de las técnicas existentes en la actualidad o incluso con técnicas que se desarrollen en el futuro. Además, se puede observar en las publicaciones científicas sobre visualización en tiempo real de modelos de terreno que la gestión de la geometría ha obtenido siempre un trato prioritario frente al texturizado. Los estudios acerca de técnicas de niveles de detalle geométricos para terreno son notablemente más numerosos que los relativos a la aplicación de texturas sobre esos modelos geométricos.

Una de las premisas de las que parte el diseño planteado es un completo desacoplamiento de las tareas de visualización y las tareas de análisis y tratamiento de la información, realizadas habitualmente en los SIG. Se entiende que esto es desde el punto de vista del desarrollador, a nivel de arquitectura de *software*. Desde el punto de vista del usuario, no tiene por qué apreciarse esta separación, sino que por el contrario podrá disponer de una interfaz con todas las funcionalidades perfectamente integradas. Sin embargo, la decisión de aislar el subsistema de visualización facilita su reutilización en diferentes sistemas.

Se propone una arquitectura cliente-servidor donde el motor de visualización y los SIG se conectan a través de una red de comunicaciones, tal y como

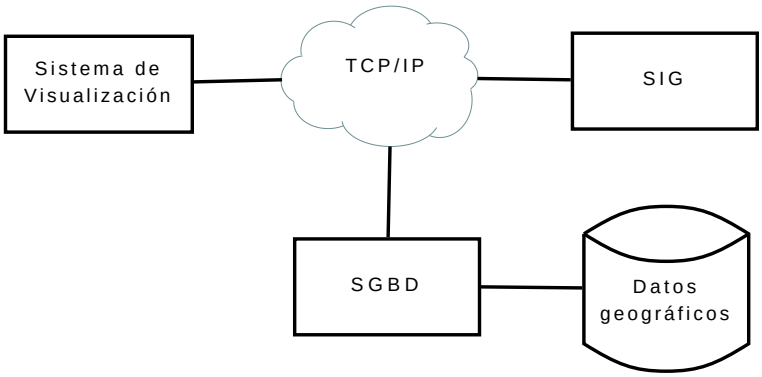


Figura 1.1: Arquitectura general del sistema

se muestra en la figura 1.1. De esta forma, siguiendo un protocolo de comunicación preestablecido, se coordinan las tareas de análisis, procesamiento, consultas, selección de capas de información, etc. con las tareas de navegación sobre el terreno 3D y visualización y selección de dicha información sobre éste.

El sistema aquí descrito se centra por lo tanto en los aspectos de visualización, delegando en aplicaciones SIG externas el resto de tareas que habitualmente desempeñan de forma muy eficaz. Esto es una diferencia importante en el planteamiento respecto a otros sistemas de visualización como Google Earth, Microsoft Virtual Earth 3D o NASA World Wind, que se limitan a visualizar un conjunto limitado de información geográfica sobre el terreno. El sistema de visualización se plantea como complementario a los SIG y para ser usado simultáneamente con éstos (figura 1.2). Para una perfecta coordinación entre ambos, se desarrollará un lenguaje o protocolo de comunicación entre ellos.

Esta comunicación es bidireccional en el sentido de que no se limita a controlar uno desde el otro. Por una parte el SIG envíará información vectorial organizada en capas al sistema de visualización, enviará las órdenes para mostrar u ocultar capas, modificará los parámetros de visualización que determinan el aspecto de la información sobre el terreno, etc. Por otra parte el usuario puede, a través de la vista 3D, solicitar información acerca de determinados elementos seleccionándolos directamente sobre el terreno. Esto enviará una consulta al GIS que la resolverá y modificará la información visualizada en 3D, creará nuevos elementos, destruirá elementos existentes, etc. También puede ser interesante en algunos casos el envío de órdenes de navegación desde el SIG al sistema de visualización. De esta forma que no será necesario que el usuario "pilote" de forma manual para localizar un lugar concreto, al cual, por otra parte, no tiene por qué saber cómo llegar.

Se comienza planteando una arquitectura genérica de visualización de terrenos que se alimenta de diversas fuentes de datos geoespaciales (figura

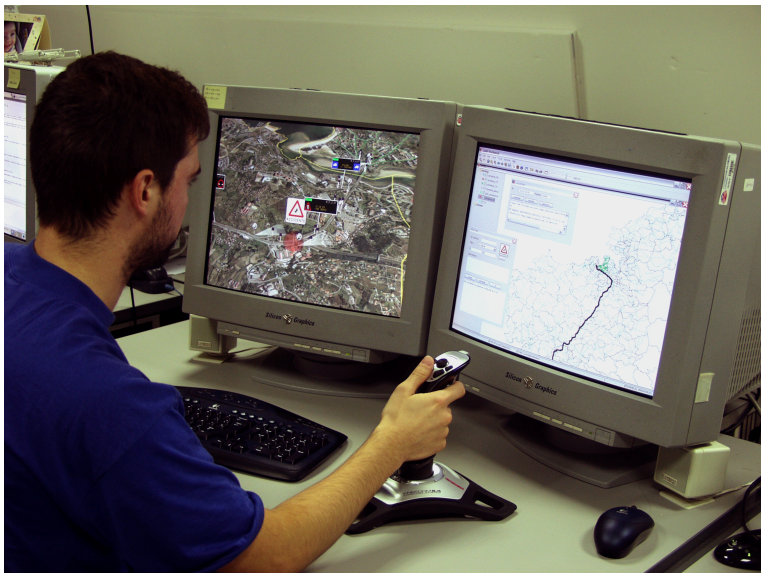


Figura 1.2: Uso combinado del sistema de visualización con un SIG utilizando múltiples pantallas

1.3). La naturaleza de estas fuentes puede ser muy diversa, incluyendo datos de elevación de terreno, imágenes de satélite y fotografía aérea, modelos 3D de construcciones y datos vectoriales 2D ubicados en la superficie del terreno. Además, algunos de estos datos serán estáticos, mientras que otros variarán dinámicamente en tiempo real.

Partiendo de esta arquitectura, nos centramos en el estudio de la aplicación de imágenes aéreas y de satélite combinadas con datos vectoriales 2D procedentes de SIG. Todos estos datos geoespaciales se procesan, se combinan y se aplican con técnicas de mapeado de texturas sobre la geometría del terreno. Se utilizará un motor de geometría [136] basado en una variación de la técnica de DeBoer [61] para las pruebas del sistema, aunque se podría utilizar cualquiera de las múltiples técnicas existentes para tal fin, por lo que no se profundizará más en este aspecto.

Una de las características más importantes del subsistema de texturizado propuesto es precisamente su completa independencia del motor de geometría, de forma que no se impone ninguna restricción al diseño ni a la implementación de ambos sistemas ni a sus correspondientes bases de datos, facilitando así su reutilización en diversas situaciones. Esta es una diferencia muy importante respecto a otros trabajos sobre texturizado en alta resolución de modelos extensos de terreno, en los que existe un fuerte acoplamiento entre la gestión de la geometría y la gestión de la textura [92, 52, 91, 65, 98, 49, 50, 107, 44, 68, 137], tal y como se describe en el capítulo 2.

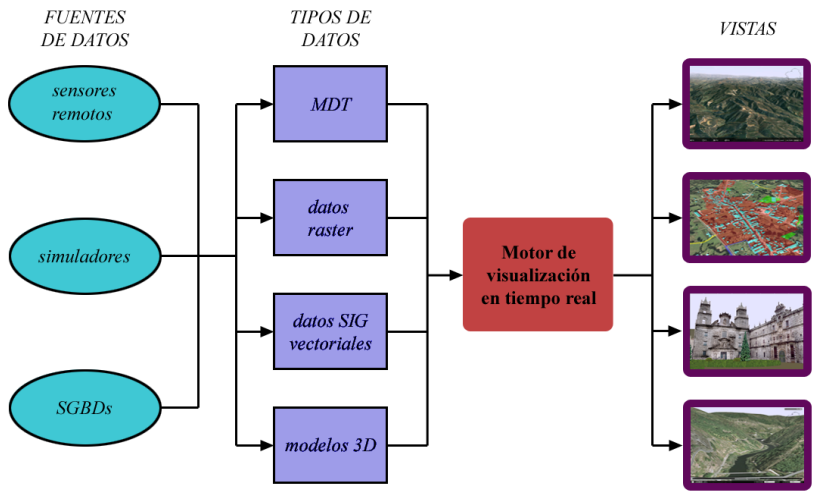


Figura 1.3: Esquema general del sistema de visualización y el acceso a los diferentes tipos de datos

1.2. La importancia de la visualización en los SIG

La importancia de la visualización 3D en los SIG ha sido puesta de manifiesto desde hace décadas. En la *IEEE Conference on Visualization* de 1994 se celebró una sesión de paneles, moderada por Theresa Marie Rhyne, donde se indicaba el retraso que sufría el campo de los SIG, en cuanto a la visualización de los datos, respecto a otros campos relacionados pero mucho más avanzados en esos aspectos, como era el de la visualización de datos científicos [124]. Reclamaba la integración de la visualización con los SIG, estableciendo tres niveles para ello: rudimentario, operativo y funcional.

A nivel rudimentario se pueden pasar datos entre ambos sistemas, pero se producen problemas por datos redundantes, proceso secuencial, interfaz dual del sistema y falta de funcionalidades para el análisis de datos multidimensionales.

A nivel operativo se facilita la transferencia de datos en tiempo real entre ambos sistemas, eliminando la redundancia de los datos. Sin embargo este nivel dicha transferencia introduce problemas de rendimiento.

La integración a nivel funcional es la deseable, permitiendo un análisis y visualización interactivas. Se extienden los sistemas de visualización adaptándolos a la información manejada por los SIG y se extienden las capacidades analíticas de los SIG a las cuatro dimensiones (espacio-tiempo), además de proporcionarse una interfaz unificada para los usuarios. El motor de visualización desarrollado en esta tesis se ha diseñado para poder ser integrado a nivel funcional con cualquier *software* lo suficientemente abierto como para permitirlo. De hecho, una de las principales barreras para conseguir alcanzar

el nivel funcional es la política privativa del *software* comercial.

Loey Knapp identificaba los tres principales problemas de los SIG como los siguientes:

- Que manejaban únicamente información 2D.
- Que la visualización se limita a vistas espaciales de los datos.
- Que la capacidad de interacción con esos datos es ínfima.

Peter Kochevar defendía la visualización como una ayuda fundamental que facilita la comprensión de los datos por parte de los humanos, y en consecuencia la adquisición del conocimiento. No se trata de un simple acto de presentación de la información, sino que va mucho más alla. La visualización debe ser un proceso bidireccional que permita al usuario interactuar con la información. Los sistemas de visualización se deben considerar como los portales a las bases de datos.

El Dr. Thomas H. Maze declaraba que la visualización es tan necesaria para la geografía como las matemáticas lo son para la física. Como bien saben los cartógrafos, un mapa no es sólo una herramienta para llegar desde un lugar hasta otro, sino que se trata de una forma de organizar el conocimiento para hacerlo comprensible.

Haciendo referencia a que los SIG se limitaban a trabajar con información en 2D, mientras los sistemas de visualización de datos científicos representaban con eficiencia conjuntos de datos multidimensionales, reclamaba que las geociencias necesitan contiguidad tanto espacial como temporal para determinar causa y efecto. Lo único que se necesita para el análisis espacial es un modelo espacio-temporal y un sujeto a estudiar. La llegada de las computadoras únicamente ha cambiado la velocidad a la que se pueden realizar los análisis.

Christopher Giertsen y Anne Lucas publicaron ese mismo año un análisis de las necesidades de los usuarios y una revisión de las técnicas para la visualización en 3D de datos SIG 2D [77].

Dividieron los usos de los SIG y la visualización en análisis exploratorio de los datos, generación de hipótesis, análisis para la confirmación de las hipótesis, toma de decisiones y difusión de información. Respecto a este último punto, se indica que cuando la audiencia tiene diferentes niveles de conocimiento sobre el tema, la vista más intuitiva y fácil de interpretar debería ser usada, y que, normalmente, ésta es la que más se aproxima a la realidad.

Consideran tres tipos de usuarios, con diferentes niveles de conocimiento y por lo tanto diferentes requerimientos de visualización: los científicos, los responsables de la toma de decisiones y el público en general. Los estilos de visualización serían, progresivamente, desde el nivel más abstracto, utilizado por los científicos para las tareas de análisis, generación de hipótesis, etc...

hasta los más realistas, que correspondería a la difusión de información al público general.

1.3. Trabajos previos

El trabajo desarrollado en esta tesis está integrado dentro del proyecto SANTI [88] (Sistema Avanzado de Navegación sobre Terreno Interactivo). Este proyecto se comenzó a desarrollar en el Grupo de Visualización Avanzada en Arquitectura, Ingeniería y Urbanismo (VideaLAB) a mediados del año 1998 con financiación de la Xunta de Galicia.

Su aplicación original fue la promoción del Camino de Santiago en el año Xacobeo 1999 por parte de la Consellería de Cultura de dicha institución. La primera versión se desarrolló en C++ utilizando IRIS Performer [127] como motor de *scene graph* sobre la arquitectura InfiniteReality [114] de Silicon Graphics Inc (SGI) utilizando el sistema operativo IRIX.

Las primeras bases de datos utilizadas en el SANTI se generaron a partir de los modelos digitales de terreno MDT-200 del Instituto Geográfico Nacional (IGN) y de las imágenes del satélite Landsat. La geometría, disponible en múltiples ficheros MDT (formato ASCII) cubriendo zonas heterogéneas y solapadas, se procesó manualmente utilizando AutoCAD y Quicksurf para generar una malla regular simplificada con información de cotas cada 200 metros. Estas mallas se exportaron a formato VRML 1.0 utilizando para ello herramientas construidas a medida en lenguaje AutoLisp. Se dividió el área de Galicia en una estructura jerárquica de dos niveles. Finalmente estos fragmentos de geometría se convirtieron al formato binario de IRIS Performer (.pfb) para visualizarlos en tiempo real dentro de la aplicación interactiva. La imagen de la textura se obtuvo escaneando las ortoimágenes de satélite del IGN, ajustándolas e igualando los colores para formar un fotomosaico con todas ellas. Finalmente se trocearon con la misma división que se aplicó a la geometría del terreno.

Tras esta primera versión, fruto de las pruebas iniciales, el proyecto resultó interesante para la Consellería de Cultura, que decidió continuar financiando su desarrollo. El cambio más importante incorporado al motor de visualización de terreno en 1998 fue la implantación de la técnica de **clipmapping** [144], recién disponible en esas fechas en la arquitectura gráfica InfiniteReality sobre la que trabajábamos. Esto supuso una mejora considerable en muchos aspectos, especialmente en cuanto a calidad de visualización, rendimiento de la aplicación y a la posibilidad de modificar el motor de geometría sin tener que preocuparnos de la organización de la base de datos de textura o del sistema de texturizado. Aún hoy en día se considera la técnica de *clipmapping* como una de las mejores para la gestión de texturas de gran tamaño aplicadas en tiempo real sobre modelos geográficos extensos, y es por este motivo que la arquitectura de texturizado planteada en

esta tesis está basada en la misma filosofía. No obstante, *clipmapping* tiene algunas limitaciones que se superan en el diseño propuesto, tal y como se detallará en los siguientes capítulos. Sin duda la mayor de estas limitaciones es la necesidad de un *hardware* gráfico específico, disponible únicamente en la arquitectura InfiniteReality de SGI. Además de eliminar esta restricción y permitir el trabajo con *hardware* de bajo coste, se añadieron nuevas capacidades no disponibles en *clipmapping*, como pueden ser la posibilidad de combinar varias texturas o la generación dinámica de la textura a partir de datos vectoriales.

Las imágenes de satélite escaneadas fueron sustituidas por otras suministradas por el Servicio de Información Territorial de Galicia (SITGA), que combinaban información en color a 30 m/pixel procedente de Landsat con información pancromática a 10 m/pixel procedente de SPOT. Esto supuso una mejora importante en la calidad de visualización del terreno. Además, aprovechando la capacidad de *clipmapping* para añadir detalle adicional en zonas concretas del área de la textura[1], se incluyó fotografía aérea ortorrectificada y georreferenciada de algunas localidades como Santiago de Compostela, A Coruña, Lugo, Ourense, Pontevedra, Vigo, Ferrol, Samos y Oseira, con una resolución de 1 m/pixel. Estas fotografías se fundieron en sus bordes con la imagen de satélite para evitar cortes bruscos y mejorar el aspecto visual del terreno. El motor de geometría se modificó para mejorar su eficiencia mediante el uso de niveles de detalle en función de la distancia a la cámara. El diseño de esta versión del SANTI fue publicado en [88], donde se describen los aspectos de gestión de la geometría del terreno, gestión de las texturas y la preparación de las bases de datos.

La primera presentación pública de este proyecto se produjo en Madrid en enero de 1999, en el recinto ferial Juan Carlos I, donde la Consellería de Cultura de la Xunta de Galicia instaló el SANTI en su *stand* en la feria internacional de turismo FITUR 99 para promocionar el Camino de Santiago, con un énfasis especial puesto que 1999 era Año Santo.

Estas primeras versiones del SANTI mostraban el recorrido del Camino de Santiago representándolo sobre la imagen de satélite como una línea de color amarillo para conseguir un buen contraste. Este trazado se *rasterizaba* previamente y se combinaba con la imagen de satélite durante la generación de la base de datos. En este sentido, la filosofía de funcionamiento era similar a la utilizada por ejemplo por Virtual Earth 3D de Microsoft, que aplica la información de carreteras, fronteras, etc. *rasterizada* sobre la imagen del terreno. Esto se puede apreciar fácilmente por el hecho de que esta información de origen vectorial es visualizada con la misma degradación de la calidad que la imagen del terreno sobre la que se aplica. Esta degradación es debida a la resolución limitada y sobre todo a los artefactos producidos por la compre-

[1] Estas zonas de detalle adicional constituyen niveles incompletos de la pirámide y son denominadas habitualmente *insets*

Figura 1.4: Vistas 3D de los entornos urbanos a nivel de suelo (1999)

sión de las imágenes. Google Earth en este sentido mejora notablemente la calidad de Virtual Earth 3D, pero el resultado es equivalente a la versión de 1999 del SANTI en el sentido de que sigue mostrando información estática.

Otra aportación importante al proyecto en el año 1999 fue la inclusión de modelos 3D de ciertas poblaciones de interés dentro del Camino de Santiago, texturizados con fotografías de los edificios reales. Una de las características más destacables era que se navegaba a escalas muy diferentes de forma continua, sin cortes ni interrupciones. Se podía por ejemplo ir volando a miles de metros de altura y recorrer todo el camino de bajada hasta aterrizar en el suelo en medio de la Plaza del Obradoiro ante la fachada principal de la Catedral de Santiago. Esto supone una diferencia de varios órdenes de magnitud en las escalas manejadas. Se pasaba de trabajar con kilómetros a trabajar con milímetros, lo cual presenta numerosos problemas técnicos, principalmente debidos a los errores de precisión al almacenar las coordenadas de los vértices de un terreno muy extenso y de los modelos 3D urbanos con niveles milimétricos. Para solucionar estos problemas fue necesario trabajar con diferentes espacios o sistemas de coordenadas, de forma que se redujese el margen de valores que se almacenaban en cada coordenada.

Las poblaciones reconstruidas en 3D fueron Santiago de Compostela, Samos, Tui, Melide, Oseira y Vilanova de Lourenzá. De todas ellas la más compleja por extensión y calidad, y dada su importancia, fue Santiago de Compostela, en la que se modeló la Catedral (exterior e interior) y las plazas y calles que la rodean (Obradoiro, Fonseca, Platerías, Quintana, Azabachería e Inmaculada) con sus edificios y monumentos, todos ellos de gran importancia histórica y cultural. En la figura 1.4 se muestran algunas vistas de las poblaciones recreadas.

Tras su presentación en FITUR, el SANTI fue exhibido ese mismo año en el Palacio de Congresos de Madrid durante la presentación de la exposición "*Camiño de Santiago Virtual*", que se celebraría en Santiago de Compostela de junio a diciembre de 1999, que ya en la primera semana superó los 8000

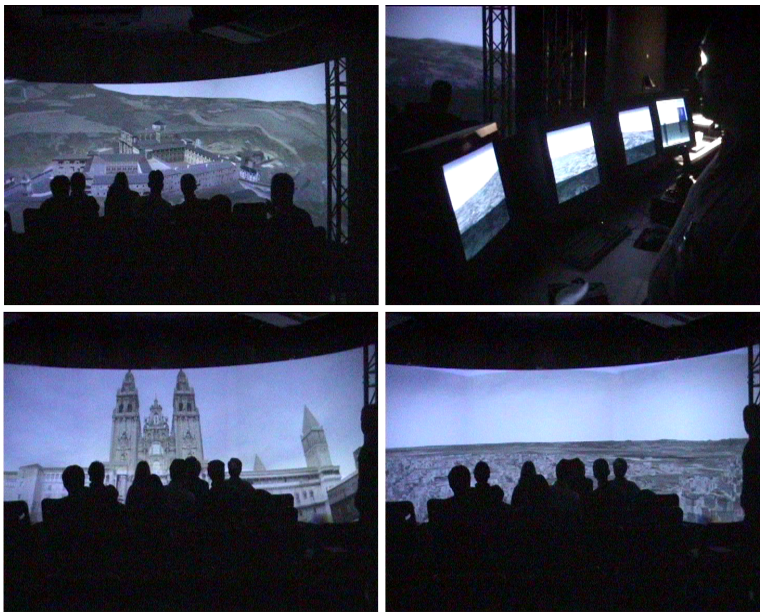


Figura 1.5: Imagina 2000 (Montecarlo). Instalación del SANTI en el Reality Center de Silicon Graphics Inc.

visitantes, llegando a recibir 300.000 en todo el año 1999.

Durante los primeros años de desarrollo del SANTI, se mantuvo contacto con el personal de ingeniería de SGI en Europa. Se realizaron colaboraciones que dieron como frutos algunas demostraciones en instalaciones especiales como los **Reality Center**. Se viajó a los centros de SGI en Reading en el Reino Unido y Cortaillod en Suiza, donde se trabajó directamente en los Reality Center para adaptar el SANTI a los sistemas con múltiples motores gráficos (*pipes*) sincronizados. El motor de visualización de terreno se preparó para poder configurarse en cualquier disposición de canales de vídeo. El montaje habitual consiste en tres proyectores Barco tritubo orientados hacia una pantalla panorámica con forma de casquete esférico que cubre un área de 120^{o} de la visión horizontal de los espectadores, obteniendo una resolución horizontal superior a los 3000 pixels. Esta instalación tiene capacidad de proyección estereoscópica activa, lo cual hace necesario que la sincronización entre los sistemas gráficos se realice a nivel de barrido (*genlock*).

Esta versión multicanal del SANTI se mostró en diferentes ferias, dentro del Reality Center instalado en el *stand* de SGI (figura 1.5). Así pues, el SANTI fue expuesto en ITEC 99 (La Haya, Holanda), I/ITSEC 99 (Orlando, USA) e Imagina 2000 (Montecarlo, Mónaco). Actualmente, esa versión del SANTI todavía está instalada en los Reality Center de las oficinas de SGI en Cortaillod, Reading y Tokio, donde se utiliza como demostrador de las capacidades gráficas de su *hardware*.

La siguiente aplicación del motor del SANTI, realizada a finales del año

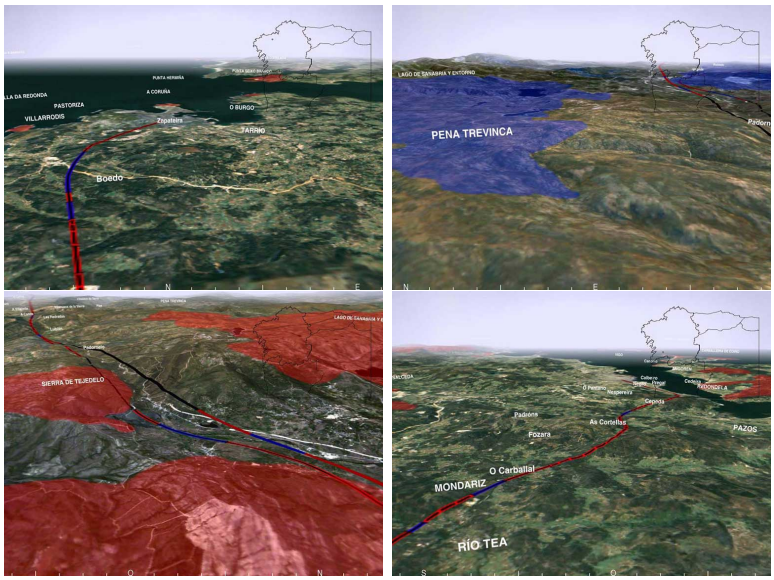


Figura 1.6: Vistas del trazado de la línea de ferrocarril de alta velocidad y representación de las zonas naturales protegidas (2000)

1999 y principios del 2000, ya no estaba relacionada con el turismo, sino con la ingeniería civil. Se visualizaron las diferentes alternativas de trazado para el tren de alta velocidad en Galicia sobre el modelo 3D del terreno. Para ello se amplió la cartografía y el modelo digital de terreno añadiendo Castilla-León. El visualizador no sólo fue utilizado por los ingenieros involucrados en el proyecto, sino que también se utilizó como herramienta de divulgación para realizar presentaciones de los proyectos de la línea de ferrocarril a la administración.

Además del trazado de las diferentes alternativas de la línea ferroviaria, se analizó su impacto ambiental, representando las diferentes zonas naturales protegidas. Para la representación de estos datos sobre el terreno, se generaron texturas específicas con las zonas y el trazado marcadas en color sobre la imagen del terreno. Cambiar estos datos suponía regenerar las texturas en un preproceso *off-line* considerablemente lento. Además, cada nueva textura suponía una ocupación de espacio en disco considerable. En la figura 1.6 se muestran algunas vistas de esta aplicación.

La siguiente ampliación del terreno representado volvió a estar aplicada al turismo y la difusión del Camino de Santiago, origen del proyecto. Se incluyó terreno de toda la zona norte de España, de forma que se podía seguir el Camino Francés desde su entrada por Roncesvalles hasta su destino final, Santiago de Compostela.

La aparición de tarjetas gráficas para PC con una buena relación prestaciones/precio nos hizo considerar la migración desde los sistemas SGI, cuyo coste de mantenimiento era excesivo. En el año 2001 se portó el *software*

del SANTI, cambiando de plataforma, tanto *hardware* como *software*. De las estaciones de trabajo de gama alta SGI Onyx2 InfiniteReality con sistema operativo IRIX, se pasó a ordenadores personales (PC), basados en sistema operativo Windows o Linux. El *hardware* utilizado en esta primera versión era un Pentium 3 con una tarjeta nVidia GeForce2. Este primer prototipo del nuevo motor se implementó directamente utilizando OpenGL, sin ninguna librería de *scene graph* por encima. Fue la primera implementación del nuevo sistema de texturizado evolucionado a partir de la técnica original de clipmapping [137] en el cual se basa el trabajo desarrollado en esta tesis doctoral, y que se describirá en detalle más adelante.

Tras este cambio de plataformas, se sustituyó de nuevo la cartografía de Galicia. La combinación de Landsat (30 m/pixel color) con SPOT (10 m/pixel pancromático) se sustituyó por una imagen de toda Galicia con una resolución de 5 m/pixel suministrada por el SITGA.

El prototipo basado en OpenGL sirvió para abandonar las costosas estaciones de trabajo gráficas de SGI, pero pronto se hizo necesario integrarlo dentro de un *scene graph* para tener una mayor versatilidad y generar de forma fácil nuevas aplicaciones integrando el terreno con otros contenidos. Se estudió el uso de OpenGL Performer (el sucesor de IRIS Performer) cuando SGI lo lanzó, primeramente en Linux y posteriormente en Windows. Sin embargo, por diversos motivos, se buscaron alternativas a este *software* y finalmente se decidió utilizar Open Scene Graph (OSG). Este *software* tenía una arquitectura y una filosofía de trabajo muy similar a las de Performer, con lo cual se aprovechó la experiencia adquirida los años anteriores y además disponía de la enorme ventaja de ser de código abierto. Esto solucionaba muchos problemas encontrados en el pasado cuando la librería tenía algún fallo y la única alternativa era esperar a que SGI lo solucionase y publicase una nueva versión o un parche para la actual.

A la hora de realizar el cambio a OSG se plantearon y se estudiaron varias alternativas. Se probó una implementación en la que la gestión del terreno se realizaba a partir de una jerarquía de nodos del *scene graph*. Sin embargo, esto suponía una sobrecarga muy grande en tiempo de proceso y sobre todo en uso de memoria. Por este motivo, finalmente se optó por encapsular la gestión del terreno en un objeto dibujable del *scene graph*, donde se realizaba una gestión específica del terreno, mucho más eficiente en todos los aspectos. Por otra parte, al estar integrado dentro de un nodo más del *scene graph*, se mantenía la versatilidad de poder combinarlo con otros elementos 3D aparte del terreno. Tras esta "refactorización" del motor de visualización de terreno, el motor de texturizado se implementó tal y como está descrito en el artículo [137].

Las prestaciones de este nuevo motor se probaron en el año 2003 con una cartografía digital de mayor calidad. De nuevo el SITGA suministró las imágenes, en este caso ortofotografías de 1 m/pixel de resolución, correspondientes a la zona de Coruña y Ferrol. Con estas imágenes de muestra

Figura 1.7: Comparativa de calidad entre bases de datos procedentes de imágenes de satélite y de ortofotografías aéreas

se desarrolló un prototipo para demostrar la diferencia de aspecto entre las bases de datos. La calidad era muy superior, no sólo por la resolución en sí, sino por el color y la calidad de la propia fotografía. Estas diferencias se pueden apreciar claramente en la figura 1.7.

Esta demostración sirvió para conseguir imágenes de toda Galicia a 0,25 m/pixel, procedentes del SIGPAC, que es la base de datos que se utiliza actualmente en el SANTI.

En el año 2004 se comenzó a plantear la necesidad de enlazar el sistema de visualización de terreno con bases de datos geográficas externas. Una de las aplicaciones que fomentó esta ampliación fue la visualización de los datos de cobertura radioeléctrica de la emisión de televisión digital terrestre (TDT) en Galicia. Se trabajó en la conexión del SANTI con el *software*

de cálculo de dicha cobertura (COBRA). Este proyecto, financiado por la Xunta de Galicia, fue el comienzo de la conexión del SANTI con SIG, que posteriormente derivaría en el presente trabajo.

En 2005 el motor basado en OSG fue adaptado para poder visualizar el terreno a través de varios canales de vídeo simultáneos, de forma similar a como se realizaba en los Reality Center de SGI. Para esto se utilizaron varios nodos de render (PCs) sincronizados a través de una red *ethernet* a 100 Mbits/s. La sincronía se realizaba a nivel de frame, el sincronizar los barridos queda sujeto a la disponibilidad de *hardware* gráfico con esa capacidad (*genlock*/*framelock*).

Posteriormente se realizó una colaboración con la empresa ORAD. Se adaptó el motor de visualización para su funcionamiento en un *cluster* de sistemas DVG (Digital Video Graphics) con las diferentes configuraciones disponibles: *antialiasing*, *time division*, *screen division*, *scene division*, etc. Esto constituyó una buena puesta a prueba para el sistema de sincronización multicanal. Se viajó al centro de desarrollo de ORAD en Szczecin (Polonia) donde se trabajó para adecuar el motor del SANTI a sus sistemas en *cluster*. El resultado de esta colaboración se utilizó como demostración de la potencia gráfica de las máquinas de ORAD en el SIGGRAPH 2006.

En resumen, desde su origen, el SANTI ha ido evolucionando a lo largo de los años a través de diversas aplicaciones en campos tan variados como el turismo, la cultura, la educación, la ingeniería o las telecomunicaciones. Tras la incorporación de la capacidad de conexión con bases de datos externas y SIGs para la visualización de datos dinámicos en tiempo real, objeto de trabajo de la presente tesis doctoral, las primeras aplicaciones planteadas son la asistencia al control de tráfico, sistemas de prevención y control de incendios y gestión y conservación de carreteras, de las cuales ya se han realizado prototipos para validar el sistema.

1.4. Aportación de la investigación

Tomando como punto de partida el trabajo previo desarrollado en el proyecto SANTI, descrito en la sección anterior, en esta tesis se propone una nueva forma de visualizar el terreno con información SIG. La información vectorial se aplicará sobre el terreno en forma de textura. De esta manera, la técnica desarrollada combina las ventajas de un sistema de gestión de imágenes multirresolución de terreno basado en la técnica de *clipmapping* [144] con la aplicación en tiempo real de información vectorial dinámica. Se plantean algunas mejoras sobre los *clipmaps* originales, como la posibilidad de combinar diferentes texturas de gran tamaño y simultáneamente su combinación con *shaders* para generar zonas de forma procedimental (por ejemplo el mar, donde no se dispone de información en la base de datos de imagen del terreno, y además se trata de un elemento con unas características visuales

diferentes al terreno, como los reflejos del sol o el oleaje).

La adquisición de los datos debe ser lo suficientemente flexible como para poder recibir la información geográfica de servidores de datos SIG, *software* SIG de análisis o cartografía digital, *software* de simulación científica, sistemas gestores de bases de datos, sistemas de captura de datos en tiempo real, sensores remotos, etc.

Por este motivo, uno de los puntos fundamentales tenidos en cuenta durante el diseño es la interoperabilidad del sistema. Este trabajo se apoya en estándares como los definidos por el OGC [15] para el intercambio de información geográfica entre el motor de visualización y otras entidades externas (principalmente *software* SIG).

Sin duda el aspecto más importante ha sido el mantener una respuesta interactiva del sistema con una fluidez absoluta, manteniendo en todo momento la mejor calidad posible. Se da prioridad a mantener una frecuencia de actualización (*frame rate*) constante de al menos 60 fotogramas por segundo (fps). Definiendo los mecanismos de control que garantizan que el tiempo de fotograma no se supere, el sistema está diseñado para obtener una calidad de visualización óptima. Dicha calidad está condicionada por el *hardware* utilizado, especialmente el sistema gráfico o GPU.

Una vez que hacemos mención de la calidad de la visualización, es necesario definir ésta en base a ciertos parámetros que la determinan y que han sido tenidos en cuenta durante el desarrollo de esta investigación como objetivos a conseguir. Además de los parámetros evidentes como las resoluciones en texels que se utilizan o la profundidad de color, existen otros aspectos no menos importantes.

El primero de ellos es la coherencia espacial, mencionada ya por Cosman en [55], uno de los trabajos pioneros y más importantes sobre el texturizado de grandes extensiones de terreno. Esta coherencia espacial es uno de los aspectos peor tratados por los sistemas disponibles en la actualidad, que gestionan las texturas de gran tamaño como un mosaico de teselas o *tiles* que aplican al terreno. Una de las principales aportaciones de esta tesis radica en esa diferencia muy importante en la filosofía del tratamiento de la textura y los datos SIG aplicados sobre ella para conservar la coherencia espacial. Estos aspectos serán tratados en profundidad en los siguientes capítulos de esta memoria.

Otra característica importante es la forma de actualización de la información aplicada sobre el terreno. Se realiza un refinamiento progresivo coherente y diseñado para minimizar el impacto visual del cambio de nivel de detalle.

Finalmente, uno de los aspectos fundamentales a trabajar en un sistema de este tipo es la reducción de artefactos visibles en la imagen, especialmente en movimiento, debidos a problemas de *aliasing*, tanto de las texturas como de los datos vectoriales procedentes de los SIG. Para ello se aplican diferentes técnicas de filtrado y *antialiasing*, estudiando especialmente el problema de la anisotropía en las texturas mapeadas sobre el terreno 3D, que está causado

por la posición de la cámara, frecuentemente enfocada hacia el horizonte.

Además de los objetivos en cuanto a calidad y rendimiento, se ha tenido especial cuidado en facilitar que el desarrollo del sistema de visualización sea absolutamente independiente de la geometría sobre la que se representan los datos, tanto *raster* como vectoriales. Por lo tanto, el sistema será aplicable de forma inmediata a cualquier motor de visualización de terreno 2D o 3D ya existente, añadiéndole la capacidad de visualizar información estática o dinámica procedente de SGBDs, SIGs o cualquier otra fuente local o remota, sin hacer bajar el *frame rate*.

Más aún, la información aplicada se puede combinar con cualquier otra a través de *shaders*, sin más restricciones que las que imponga el *hardware* utilizado. Esto posibilita la combinación de la información geográfica con todo tipo de materiales, animaciones, técnicas de iluminación y efectos especiales, posibilitando el surgimiento de nuevas aplicaciones y un nuevo concepto de visualización de la información geoespacial.

1.5. Estructura de la memoria

El resto de este documento se ha estructurado de la siguiente manera.

En el capítulo 2 se exponen los conceptos fundamentales y se hace un análisis de los trabajos publicados relacionados con el tema de estudio de esta tesis, incluyendo los SIG, la visualización de terreno (tanto modelos topográficos como sistemas de texturizado) y la visualización de datos vectoriales sobre el terreno 3D.

El capítulo 3 establece el planteamiento de este trabajo de investigación, el punto de partida que se toma, los objetivos que se pretenden conseguir y la metodología aplicada para conseguir esos objetivos.

En el capítulo 4 se describe en detalle el sistema de visualización desarrollado en esta investigación, analizando desde la arquitectura general hasta los detalles de diseño de los módulos más relevantes.

El capítulo 5 detalla las pruebas realizadas sobre el sistema construido y analiza los resultados obtenidos en dichas pruebas.

En el capítulo 6 se presentan las conclusiones obtenidas tras la finalización de la investigación y se exponen las futuras líneas de trabajo que surgen a partir de los resultados obtenidos.

Los apéndices A a J completan y amplían los contenidos de los capítulos 4 (desarrollo) y 5 (resultados).

Finalmente se incluye la bibliografía consultada para la realización de esta tesis y que figura referenciada a lo largo del texto.

Capítulo 2

Antecedentes

2.1. Introducción histórica

La **cartografía** tiene su origen hace varios miles de años, fruto de la necesidad o curiosidad que el hombre ha sentido desde tiempos remotos por conocer el entorno en el que vive. Los primeros mapas que se conocen, hallados en el complejo de cuevas de Lascaux, situado al suroeste de Francia, están fechados en el año 16.500 a.C. Se cree que se trata de mapas de estrellas y constelaciones. Por lo tanto, supuestamente, los primeros mapas corresponden al cielo y no a la Tierra.

Tras estas pinturas rupestres, el primer "mapa terrestre" hallado se sitúa en torno al año 6.200 a.C. [112] en la ciudad de Çatalhöyük, en Anatolia (Turquía). Se trata del plano de una ciudad representado sobre un muro.

Existen diversos mapas en tablillas de barro de la antigua Babilonia, que corresponden a escalas muy variadas, desde aproximaciones de mapas del mundo hasta mapas locales con detalles de edificios. Uno de los que se consideran más antiguos ejemplares de estos mapas de Babilonia está datado por diferentes autores entre el año 2.300 y el 3.800 a.C.

Tras estos primeros mapas, el interés por la cartografía fue creciendo, así como la complejidad y la escala. Partiendo del entorno inmediato, el territorio conocido se fue extendiendo, principalmente potenciado por las rutas de comercio. A medida que el área estudiada aumenta, entra en juego la forma y el tamaño de nuestro planeta. A la hora de ubicar posiciones, tanto relativas como absolutas, medir distancias, etc. en un área extensa, necesitamos tener en cuenta la curvatura de la Tierra. Entramos entonces en el campo de la **geodesia**, que literalmente significa "dividir la Tierra".

La geodesia es la ciencia que estudia la forma y el tamaño de la Tierra, así como la manera de localizar elementos en su superficie. Al igual que la cartografía, la geodesia no es un campo moderno sino que sus orígenes se remontan a la antigüedad. Ya en tiempos de la Grecia Clásica, Pitagoras (580-500 a.C.) consideraba la Tierra como una esfera (quizás porque la esfera

era la forma perfecta según los filósofos griegos). Esto contrastaba con la consideración de la Tierra como una superficie plana, que partiendo de un desconocimiento inicial podría ser la más plausible (Homero consideraba una figura en forma de disco y Anaxímenes en forma de rectángulo). Soportando la teoría de la forma esférica, que era la más aceptada en la época de los filósofos griegos, Aristóteles (384-322 a.C.) (y posteriormente Arquímedes) realizó las primeras estimaciones del radio de la Tierra con métodos que no se conocen. Fue Eratóstenes (275-195 a.C.), el bibliotecario de la famosa biblioteca de Alejandría, en el año 230 a.C., el primero que realizó esta tarea con un método científico conocido.

Fue también Eratóstenes quien acuñó el término **geografía** para referirse al estudio de la Tierra y todos los elementos que existen en ella, incluidos por supuesto los seres humanos. La geografía por tanto es la ciencia que engloba muchas de las otras que afectan a este proyecto, como pueden ser la cartografía, la geodesia y la topografía, estudiando la distribución en el espacio y el tiempo de fenómenos, procesos y características, ya sean naturales o producto de la acción del hombre.

El trabajo de Eratóstentes fue posteriormente criticado por Hiparcos (190-120 a.C.), quien creó el sistema de coordenadas geográficas utilizado en la actualidad, que divide el planeta utilizando meridianos y paralelos y permite referenciar cualquier punto sobre la superficie del planeta a través de sus dos coordenadas angulares, longitud (λ) y latitud (ϕ), utilizando el sistema sexagesimal, que divide la circunferencia en 360^{o}, cada grado en 60' y cada minuto en 60".

El desarrollo de todas estas ciencias contribuyó a la mejora del conocimiento acerca del planeta que habitamos y posibilitó los grandes viajes alrededor del mundo y el descubrimiento de nuevas tierras. Así pues, los grandes viajeros y descubridores como Cristobal Colón, Marco Polo, Juan Sebastián Elcano, Fernando de Magallanes o John Cook, fueron grandes impulsores de las ciencias relacionadas con la geografía, especialmente la cartografía y la geodesia, fundamentales para la navegación marítima.

Actualmente, puede decirse que pocos lugares del planeta quedan por explorar, si es que todavía existe alguno. Mucho ha cambiado el conocimiento de nuestro entorno desde las primeras discusiones de los griegos acerca de la forma esférica de la Tierra o desde las tablillas de barro de la antigua Babilonia.

En la actual era de la información y las comunicaciones, con la tecnología disponible, el estudio de los aspectos geográficos se puede realizar con una gran precisión y de forma prácticamente inmediata. Se dispone por ejemplo de satélites que capturan información acerca de la Tierra de forma continua desde el espacio. Los satélites nos sirven también para ubicar la posición en cualquier punto de la superficie de la Tierra de forma inmediata y con una precisión cercana a un metro, a través de sistemas como el **NAVSTAR GPS** (*Navigation Satellite Timing And Ranging Global Positioning System*)

americano o el futuro **Galileo**, desarrollado por la Unión Europea junto con otros paises de Asia y África, cuya entrada en funcionamiento está prevista para el año 2010.

La cartografía ha evolucionado a nuevas formas gracias a los avances tecnológicos. Así pues, la cartografía tradicional está siendo sustituída por la denominada **cartografía digital**, concepto muy ligado a los **sistemas de información geográfica** basados en ordenadores que trataremos más adelante en la sección 2.2.

Estas diferentes aplicaciones de la informática al servicio de la geografía se engloban en la reciente disciplina denominada **geomática** [100], término acuñado en 1969 por B. Dubuisson. La geomática es, en esencia, el tratamiento automatizado de la información geográfica, todo lo que tenga que ver con la captura, almacenamiento, procesamiento y transmisión de información referenciada espacialmente. Dentro de la geomática se suelen incluir todo tipo de aspectos relacionados con la geografía donde se aplique el uso de computadoras, como la geodesia, cartografía, fotogrametría, SIG, GPS, reconocimiento del terreno, sensores remotos, etc. En algunos casos se diferencia la **geotelemática** como aspecto independiente enfocado en la transmisión de los datos geoespaciales.

2.2. Sistemas de información geográfica (SIG)

Los **Sistemas de Información Geográfica** o SIG[1] son la evolución de la cartografía tradicional en combinación con la informática, especialmente el campo de las **bases de datos** aplicado a la información espacial.

El término "geográfico" hace referencia a nuestro planeta[2]. Es habitual hablar de **información espacial** de forma equivalente, aunque en este caso el término es más amplio y no se limita a la Tierra. Recientemente se ha hecho frecuente el uso del término **geoespacial** para hacer referencia al subconjunto de la información espacial referida a la superficie del globo terrestre[3]. En gran parte de los casos, los conceptos y técnicas utilizados en los SIG son extrapolables a información espacial no referida a la Tierra, como pueden ser otros planetas, el cosmos o incluso otros ámbitos como el espacio del cuerpo humano a través de imágenes médicas [106]. Una prueba del auge de este término es el hecho de que el anteriormente denominado **OpenGIS Consortium** ha sido renombrado en septiembre de 2004 a **Open Geospatial Consortium** [15].

[1] Aunque el acrónimo español es SIG, es muy frecuente el uso del acrónimo inglés **GIS** por *Geographic Information System*. En este texto se ha optado por la primera de las opciones.

[2] Su raíz etimológica es la palabra griega **gaea** ($\gamma\alpha\iota\alpha$) que significa Tierra.

[3] Aunque aquí nos referimos a la superficie, también se podría trabajar con datos acerca del subsuelo y de la atmósfera en los SIG, identificando su altura respecto a un nivel de referencia.

Figura 2.1: Mapa realizado por John Snow para ubicar los casos del brote de cólera surgido en el Soho londinense en 1854

La historia de los SIG como sistemas informáticos se remonta a la década de los 60. Sin embargo, el tipo de técnicas utilizadas en los SIG ya se utilizaba anteriormente. Uno de los primeros usos de las técnicas de información geográfica documentados es un mapa realizado por el Dr. John Snow en 1854. Este mapa, mostrado en la figura 2.1, ubicaba a través de puntos los casos de cólera detectados en el área del Soho en Londres y permitió, estudiando la distribución de estos casos, detectar el origen de la enfermedad en una bomba de agua situada en el centro de la zona afectada.

El **Canada Geographic Information System** (CGIS) es considerado como el primer SIG de la historia. Fue desarrollado en Ottawa en 1964 por el gobierno canadiense con el objetivo inicial de crear un inventario de los recursos existentes en el territorio de ese país e identificar sus posibles usos. CGIS fue concebido como una herramienta de medición más que de creación de mapas y por lo tanto trataba la información en forma de tablas. Permitía analizar la información trabajando en capas, disponía de herramientas de medición y permitía el análisis de complejos conjuntos de datos en extensiones tan amplias como un continente completo.

Este sistema pionero fue seguido de otras iniciativas en EEUU (US Bureau of the Census, DIME program, ODYSSEY GIS) y Reino Unido (ECU) hacia finales de la década de los 60. El desarrollo de los SIG fue potenciado por el uso en esa misma época de satélites para capturar datos de la Tierra de forma remota (*remote sensing*), especialmente con el surgimiento de los satélites de uso civil, como Landsat a principios de los 70.

Sin embargo, fue en los años 80 cuando comenzó el verdadero crecimiento y expansión de los SIG, debido al abaratamiento de los equipos informáticos. Es en estas fechas donde, según algunos autores [106], comienza la era de la comercialización, que sigue a la era de la innovación y precede a la era de la explotación, que comenzará en 1999 con el lanzamiento del satélite IKONOS, capaz de obtener imágenes de la superficie de la Tierra con una resolución de 90cm. A mediados de los 80 fue también cuando comenzó a estar operativo el GPS.

Otro hito importante en la historia de los SIG es el nacimiento del OpenGIS Consortium, posteriormente renombrado Open Geospatial Consortium (OGC). Se trata de un organismo internacional formado por más de 300 empresas, organizaciones gubernamentales, universidades y en general usuarios y proveedores de servicios geoespaciales. Su objetivo principal es la mejora de la interoperabilidad entre este tipo de sistemas a través del desarrollo de especificaciones de interfaces abiertos y públicos. El OGC trabaja en colaboración con el comité técnico de la organización internacional de estandarización ISO/TC211 (*Geographic Information/Geomatics*), encargado de definir la familia de estándares ISO 19100, que se enfocan en el campo de la información geográfica o la geomática. En teoría el ISO/TC211 desarrolla los estándares abstractos y el OGC se encarga del nivel de implementación. Sin embargo, en la práctica la línea de separación entre ellos está bastante difusa [100].

Puesto que la interoperabilidad del sistema de visualización con cualquier SIG es uno de los objetivos de este trabajo, se ha mostrado un especial interés en seguir las recomendaciones del OGC y cumplir las interfaces por él publicadas.

2.2.1. La información geográfica

Las aplicaciones de los SIG son muy variadas. Las más habituales incluyen el planeamiento urbano, demografía, gestión de recursos naturales, hidrología, geología, planificación de estrategias comerciales, planificación de rutas, etc. Cada una de ellas tiene sus propias particularidades y trabaja con información diferente. Sin embargo hay una serie de aspectos comunes a toda la información manejada por los SIG, y que son precisamente los que caracterizan a este tipo de sistemas.

En primer lugar, se pueden distinguir dos tipos de datos de naturalezas diferentes aunque complementarios: los **datos espaciales** y los datos no espaciales o **atributos** de los datos espaciales. Pensemos por ejemplo en un conjunto de información sobre establecimientos hoteleros en una zona determinada. La ubicación de cada elemento de este conjunto constituye la información espacial, que consistirá, en este caso, en un punto sobre el mapa con sus coordenadas en el sistema de coordenadas utilizado. Además, cada uno de estos establecimientos hoteleros, cuya posición en el mapa nos viene

indicada por los datos espaciales, tendrá asociados una serie de datos adicionales, como el nombre, tipo de establecimiento, servicios disponibles o número de habitaciones libres. Esta información no tiene explícitamente la componente espacial, puesto que ésta se obtiene a través de los datos espaciales del elemento al que va asociada. Los atributos se suelen almacenar en tablas de bases de datos relacionales, y no tienen por qué limitarse a información textual, sino que también se podrían incluir otros tipos de información, como fotografías, vídeo, audio, etc.

Con el enfoque puesto en los datos espaciales, se puede considerar otra división en dos tipos de representación: *raster*[4] y vectorial.

En el caso de la **información *raster***, se discretiza un espacio continuo, como es el terreno, en celdas de idéntico tamaño y se almacena una información determinada asociada a cada una de estas celdas. Esta información almacenada por celda puede ser de diferentes tipos, como mediciones de radiaciones en cualquier banda del espectro electromagnético (ya sean bandas de color dentro del espectro visible o cualquier otra como en el caso de los sensores de infrarrojos o sistemas de radar) o datos temáticos acerca del terreno (usos del suelo, densidad de población, temperatura, tipo de vegetación, etc). En la mayoría de los casos, la información temática asociada a las celdas se convierte a información de color utilizando una escala predefinida para poder visualizarla en pantalla. La figura 2.2 ilustra algunos ejemplos de información *raster*.

La **información vectorial** está formada por primitivas geométricas, con las posiciones de sus vértices en el sistema de coordenadas utilizado. Generalmente se trata de **puntos**, **líneas** y **polígonos**, o variaciones de estos (las líneas se pueden combinar en polilíneas, los polígonos pueden definir un área más compleja que incluya "huecos" determinados por otros polígonos, etc). La figura 2.3 ilustra algunos ejemplos de información vectorial.

Estos datos, ya sea en formato *raster* o vectorial, pueden almacenar una información **continua** o **discreta**. La información continua puede interpolarse a partir de las muestras disponibles, mientras que en el caso de la información discreta no tiene sentido interpolar entre los elementos existentes. Ejemplos del primer caso son la distribución sobre el terreno de parámetros como altura, temperatura, pluviometría o presión atmosférica. Un ejemplo de información discreta pueden ser los elementos de un inventario de infraestructuras (como depósitos de agua, edificios, aeropuertos), localización de ciudades, trazados de viales (datos vectoriales) o usos del suelo (datos *raster*). En algunos casos, la información continua se discretiza en una escala reducida de posibles valores para facilitar su análisis y su visualización.

Según la información a representar será más adecuado un sistema u otro. Generalmente, para la información continua se suelen utilizar formatos

[4] La familia de estándares geográficos ISO 19100 sugiere el uso del término más general "datos reticulares" (*gridded data*) en lugar de *raster* [100].

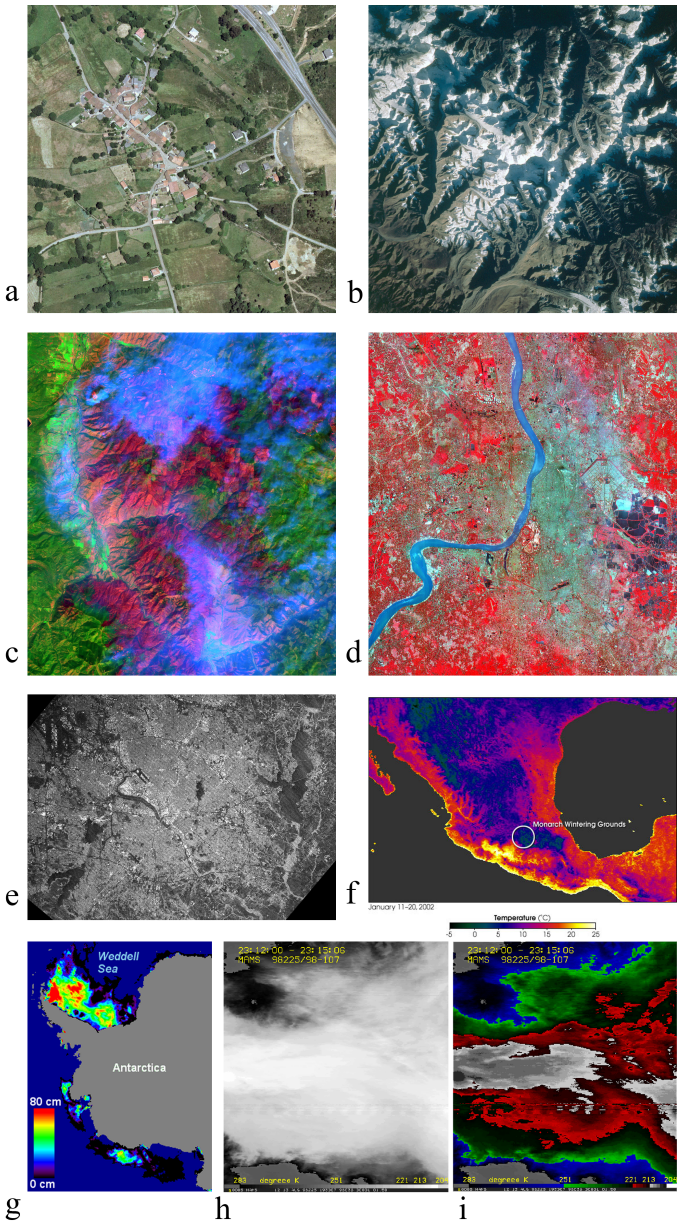


Figura 2.2: Ejemplos de información SIG continua en formato *raster*. a) Fotografía aérea. b) Imagen de satélite (espectro visible). c) Imagen de satélite multiespectral (LANDSAT 7). d) Imagen de sensor infrarrojo. e) Imagen de radar. f) Mapa temático de temperaturas. g) Mapa de profundidad de nieve en la Antártida h) Mapa de temperatura atmosférica (monocromático) i) Mapa de temperatura atmosférica (pseudocolor) (Fuentes de las imágenes: SIGPAC, NASA/GSFC/METI/ERSDAC/JAROS, y U.S./Japan ASTER Science Team)

- Información Geográfica
  - I. Espacial
    - Raster
    - Vectorial
      - Punto
      - Línea
      - Polígono
  - Atributos
    - Continua
    - Discreta

Cuadro 2.1: Clasificación de la información geográfica según sus características.

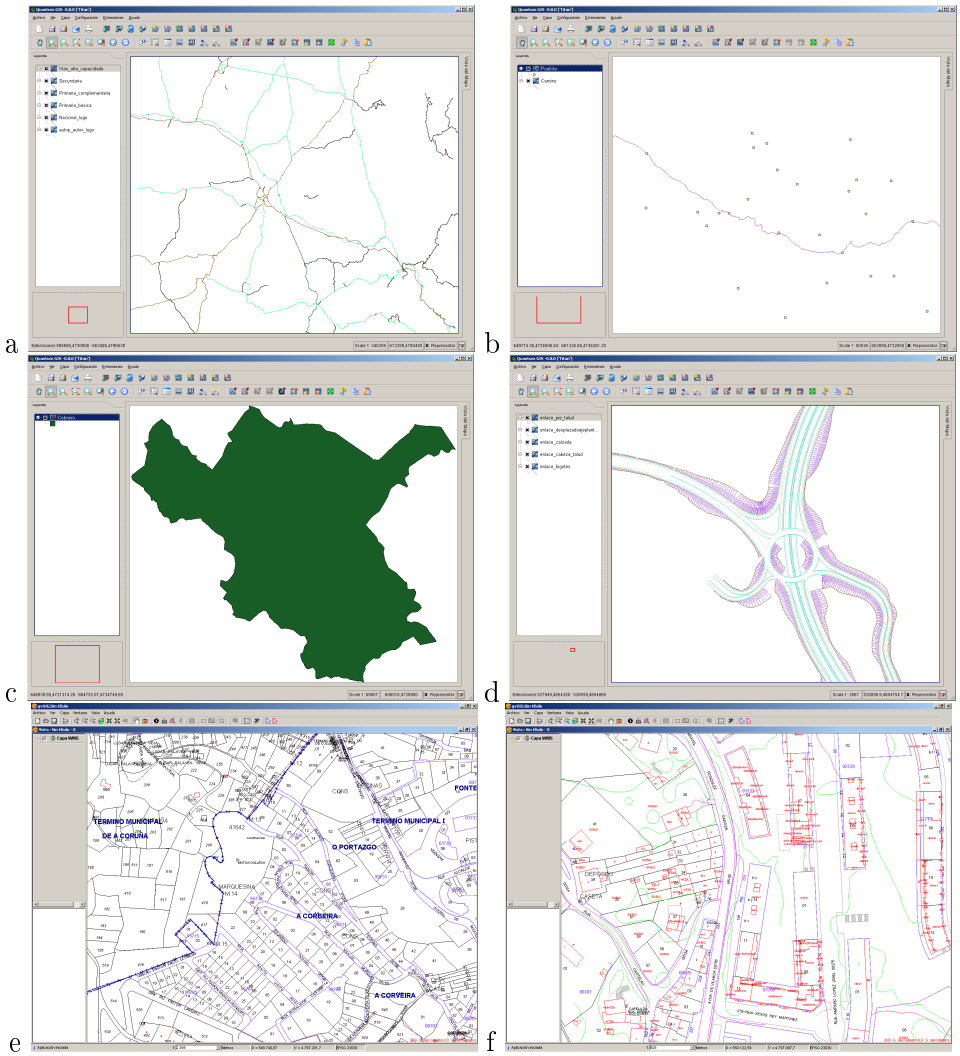


Figura 2.3: Ejemplos de información vectorial manejada por un SIG. a) Red de carreteras. b) Poblaciones de interés y ruta turística c) Área de un municipio d) Trazado del enlace de una vía rápida. e) y f) Información catastral.

raster, y para la información discreta formatos vectoriales o *raster* según el caso, principalmente en función de cómo estén distribuidos los datos. Por ejemplo, los sensores remotos utilizados en los satélites o aviones de reconocimiento, capturan la información en formato *raster* debido a las características de los propios sensores y su forma de trabajar. En cambio, por ejemplo la delimitación de parcelas del catastro se suele realizar en formato vectorial.

No obstante, el formato en que se obtengan los datos no es definitivo en el sentido de que se pueden convertir de uno a otro. Los datos de tipo *raster* pueden ser transformados en datos vectoriales mediante el proceso denominado **vectorización** y que puede realizarse de forma manual o automática. De la misma manera, los datos vectoriales se pueden muestrear en base a una división del espacio en celdas y convertir a formato *raster* en el proceso de **rasterización**.

Desde el punto de vista de la visualización, los formatos vectoriales tienen la ventaja de que se puede ampliar y reducir su tamaño en pantalla muy fácilmente y sin perder calidad. Los formatos *raster* en cambio muestran un aspecto "cuadriculado" (o en el mejor de los casos borroso) cuando se amplían de forma que las celdas tengan un tamaño considerable en pantalla. También surgen artefactos en la visualización al reducir el tamaño de celda por debajo del tamaño de pixel. Estos fenómenos denominados *aliasing* y la forma de eliminarlos o reducirlos serán explicados en detalle en la sección 2.6.1. Otra desventaja del formato *raster* es que se ocupa un mayor volumen de almacenamiento, tanto en disco como en memoria. Sin embargo, resultan muy adecuados para contener información continua, especialmente aquella capturada a partir de sensores que barren el terreno con una resolución fija.

2.2.2. Sistemas de coordenadas

A la hora de establecer la posición sobre la Tierra de los datos geográficos, ya sean las celdas en el caso de los formatos *raster* o los vértices en formatos vectoriales, se debe trabajar con un sistema de coordenadas en el que representar dichas posiciones.

Existen varias alternativas a la hora de elegir el sistema de coordenadas a utilizar. Según el estándar ISO 19111 (*Spatial referencing by coordinates*), se diferencian tres sistemas de coordenadas [100]: **cartesiano**, **elipsoidal** y **proyectado**. Este último caso se describe en detalle en la sección 2.2.3.

El sistema cartesiano, que habitualmente también se denomina **geocéntrico** establece un espacio trdimiensional donde el origen de coordenadas se sitúa en el centro de la Tierra, el eje Z se sitúa hacia el polo norte y los ejes X e Y formando el plano del ecuador, apuntando el eje X hacia el meridiano de referencia (ver figura 2.4).

En un sistema elipsoidal, las coordenadas utilizadas son la longitud (λ), latitud (ϕ) y altitud (h), tal y como se representan en la figura 2.4. Gene-

ralmente se hace referencia a las coordenadas angulares longitud y latitud como **coordenadas geográficas** o **coordenadas geodésicas**.

Puesto que los datos con los que se trabajará provienen de diversas fuentes, habrán sido generados a través de diferentes métodos, trabajando a diferentes escalas y con referencias y sistemas de representación distintos. Por este motivo, resulta fundamental trabajar con absoluta precisión en el posicionamiento de la información y conocer exactamente los sistemas de coordenadas utilizados. Sólo de esta manera se podrá ubicar correctamente la información geográfica y se posibilitará la conversión de unos sistemas a otros, de forma que toda esa información, sin importar cuán variada sea en su origen y tratamiento, encajará perfectamente en el SIG.

Si esta exactitud en la correspondencia de la posición de diferentes colecciones de información geográfica es fundamental para su correcta visualización, mucho más crítica todavía resulta para las tareas de análisis que se realicen sobre dicha información.

Por este motivo, en primer lugar se necesita un modelo de la Tierra al cual se referirán las coordenadas indicadas en la información geoespacial. En este aspecto nos servimos de la geodesia, ciencia que estudia la forma del planeta. La geodesia nos proporciona modelos sólidos regulares que aproximan la superficie de la Tierra [140]. Utilizando diferentes modelos se obtendrán diferentes valores para las coordenadas geográficas (longitud, latitud y elevación) de un mismo punto.

La topografía del planeta es completamente irregular, las zonas con una superficie más regular son los océanos. Por este motivo se toma como referencia el nivel del mar. Sin embargo, el nivel del mar no tiene una posición fija, sino que oscila a lo largo del día y del año por efecto de las mareas debidas a la influencia gravitatoria del sol y la luna sobre las masas de agua. Por lo tanto, cuando nos referimos al nivel del mar, hacemos referencia al nivel medio del mar en un punto determinado.

La aproximación matemática más sencilla que se puede utilizar es una esfera perfecta, pero esta forma no se ajusta a la compleja forma real del planeta lo suficiente como para conseguir unos resultados satisfactorios en cuanto a la precisión en las posiciones. Puesto que la Tierra está más achatada en los polos por efecto de la rotación sobre su eje, una aproximación mejor que la anterior consiste en utilizar un **elipsoide de revolución** en lugar de una esfera.

Un elipsoide de revolución (figura 2.4) se puede definir a partir de su **semieje mayor** (a) o **radio ecuatorial** y su **semieje menor** (b) o **radio polar**. Es habitual que se defina el modelo en función de su semieje mayor y de un parámetro denominado **aplanamiento** (f), que relaciona ambos ejes según la ecuación 2.1, o bien de su inverso (**aplanamiento recíproco**: $1/f$).

$$f = \frac{a - b}{a} \tag{2.1}$$

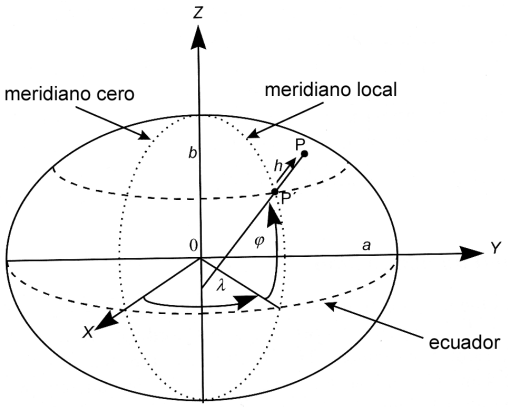


Figura 2.4: Elipsoide y sistema de coordenadas geográficas (λ,ϕ,h) y cartesianas (X,Y,Z).

Elipsoide	Año	a (m)	$1/f$	Uso
Everest	1830	6.377.276	300,802	India, Pakistán, Burma
Bessel	1841	6.377.397	299,153	Japón, Alemania, Holanda, NE China, Indonesia
WGS72	1972	6.378.135	298,26	Global
Internacional	1980	6.378.137	298.257	Buena adaptación al geoide
WGS84	1984	6.378.137	298,257223563	Global

Cuadro 2.2: Modelos de elipsoide más importantes.

El problema de estos modelos del globo terrestre es que los servicios geográficos de los diferentes países y regiones han trabajado diversos modelos concretos que resultan adecuados para la zona que estudian, pero pueden no resultar tan adecuados para otras zonas del planeta. Por este motivo, hoy en día existen numerosos modelos de elipsoides en uso (ver tabla 2.2).

Las observaciones desde el espacio en las últimas décadas, mediante el uso de satélites artificiales, han ampliado el conocimiento sobre la forma de la Tierra posibilitando la creación de un modelo más adecuado a la realidad. Se trata de una superficie equipotencial, denominada **geoide**, que se caracteriza por ser perpendicular a la dirección de la gravedad en todos sus puntos. El geoide coincide aproximadamente con el nivel medio del mar. Se trata de una superficie muy irregular y los modelos matemáticos que intentan aproximarla son extremadamente complejos[5]. Las variaciones del geoide sobre el elipsoide son de aproximadamente 100 metros arriba o abajo, por lo que para muchos usos el elipsoide es una aproximación suficientemente precisa.

[5]Como ejemplo, se puede mencionar que en EEUU la *Defense Mapping Agency* ha desarrollado un modelo matemático de grado 180 y 32.755 coeficientes.

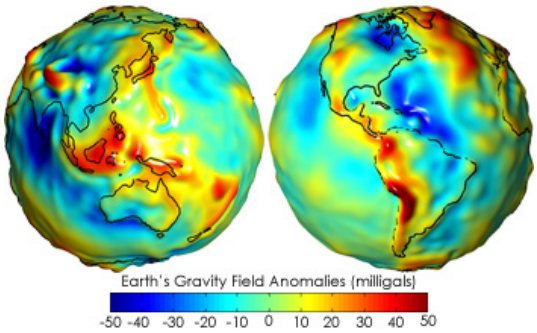


Figura 2.5: Representación del geoide, el código de colores muestra las variaciones en la gravedad (fuente: NASA)

Datum	Latitud	Longitud	Elipsoide	Uso
NAD83	39º 13' 26,686"	261º 27' 29,494"	Internacional	América
ED	52º 22' 51,45"	13º 03' 58,74"	Internacional	Europa y África
Tokyo	35º 39' 17,51"	139º 44' 40,50"	Bessel	Este de Asia
India	24º 07' 11,26"	77º 39' 12,57"	Everest	Sureste de Asia

Cuadro 2.3: *Data* geodésicos.

Para indicar una posición en la Tierra, se necesita un sistema de referencia que viene definido con lo que se conoce como **datum geodésico**. Sobre este sistema de referencia se indicarán las coordenadas del punto en cuestión. En geodesia se consideran dos tipos de *data*: el **datum horizontal** y el **datum vertical**.

El *datum* horizontal se suele componer de un modelo de elipsoide determinado (definido por un radio y su aplanamiento), un punto inicial (origen), el acimut de la orientación inicial y la separación geoidal (diferencia entre geoide y elipsoide) en el origen. Los *data* utilizados para las observaciones desde satélite son diferentes (más complejos) al que acabamos de describir. Sin embargo suelen incluir también un elipsoide de referencia para poder convertir las posiciones a coordenadas geográficas (latitud y longitud).

Los *data* más habituales según la zona, son **NAD83** (*North American Datum*, 1983), **ED** (*European Datum*), **Tokyo Datum** e **Indian Datum**, descritos en la tabla 2.3.

Respecto al *datum* vertical, éste provee un marco de referencia para los datos de elevación. Diferentes zonas del mundo tienen su propio "nivel cero" de referencia para medir las alturas. En algunos casos están asociadas al nivel medio del mar en una zona determinada y en otros casos no tienen nada que ver en absoluto, son puntos establecidos de forma arbitraria. Es importante tener en cuenta que las alturas medidas en los estudios topográficos corresponden al geoide, puesto que los instrumentos de medición están afectados por la fuerza de gravedad en la zona de la medición.

Para aplicaciones limitadas a un único continente o a áreas más pequeñas, los *data* anteriormente descritos podrían ser adecuados. Sin embargo, las aplicaciones militares que requerían trabajar en extensiones intercontinentales (principalmente balística de misiles) no disponían de un *datum* que se adecuase a todo el planeta. Por este motivo, el departamento de defensa de los EEUU comenzó a desarrollar el denominado **World Geodetic System** (WGS), con el objetivo de disponer de un modelo global que se adapte lo mejor posible a todos los continentes y que sirva para unificar todas las redes geodésicas y los diferentes sistemas de coordenadas a lo largo del planeta. El primer resultado de esta iniciativa se conoce como U.S. Department of Defense World Geodetic System 1960 (DOD WGS 60). Posteriormente se fue refinando el modelo dando como resultado los WGS 72 y el actual WGS84 (prácticamente equivalente al NAD83). Ambos sistemas están definidos como geocéntricos, aunque tienen una desviación de unos dos metros entre el centro del elipsoide y el centro de la Tierra.

Existen otros sistemas geodésicos de referencia, como el *European Terrestrial Reference Frame* de 1989 (ETRF89), el principal *datum* para Europa, desarrollado a partir de la red geodésica europea EUREF (*European Reference Frame*). Es muy similar al WGS84, con diferencias inferiores al metro.

2.2.3. Proyecciones cartográficas

Con las coordenadas geográficas (latitud, longitud y altitud) descritas en la sección anterior, podemos ubicar cualquier posición sobre el terreno respecto a un elipsoide determinado y con un *datum* concreto. El siguiente problema a resolver es cómo plasmar esta información, referente a un elipsoide, sobre un plano, que es el medio que se suele utilizar para mostrar esta información gráficamente, ya sea en papel o en una pantalla de ordenador. Esta operación de construcción de mapas se denomina **proyección cartográfica**. Ptolomeo (100-178) fue uno de los pioneros, realizando en el año 150 un mapa de la geografía conocida en la época del Imperio Romano (figura 2.6).

La proyección cartográfica consiste en una correspondencia biunívoca entre los puntos de la superficie terrestre y el llamado plano de proyección. Por lo tanto, define una operación que convierte las coordenadas geográficas en las coordenadas cartesianas del plano.

$$(\lambda, \phi) \rightarrow (x, y) \tag{2.2}$$

Al realizar una proyección de la geografía, de forma aproximadamente esférica, sobre un plano, es inevitable que se produzcan ciertas distorsiones. Estas distorsiones serán tanto mayores cuanto más grande sea la zona representada. Existen tres tipos de distorsión que afectan a otras tantas propiedades geométricas del terreno: los ángulos, las superficies y las distancias.

Figura 2.6: Mapa de la geografía conocida en la época del Imperio Romano, realizado por Ptolomeo en el año 150 (reconstruido en el siglo XV).

Según el tipo de proyección elegido, se conservarán unas u otras propiedades, pero es imposible conservarlas todas simultáneamente. La propiedad de conservar los ángulos en el terreno se denomina **conformidad**, la propiedad de conservar las superficies (áreas) se llama **equivalencia** y la propiedad de conservar las distancias entre puntos se denomina **equidistancia**.

Existen tres posibles superficies de proyección: el plano, el cilindro y el cono (ver figura 2.7), que dan lugar a las llamadas proyecciones **planas** (o **acimutales**), **cilíndricas** y **cónicas** respectivamente. En los dos últimos casos se dice que son superficies desarrollables, puesto que se pueden desarrollar sobre un plano.

En las cilíndricas y cónicas se toma el centro de la Tierra como punto de vista para la proyección. En el caso de las proyecciones acimutales, el punto de vista se pude situar en el infinito, en un punto exterior a la Tierra, en las antípodas del punto de tangencia del plano de proyección o en el centro de la esfera, dando lugar a una proyección **ortográfica**, **escenográfica**, **estereográfica** o **gnomónica** respectivamente (ver figura 2.8).

Uno de los sistemas más utilizados es la proyección **UTM** (*Universal Transverse Mercator*). Es una proyección cilíndrica transversa (figura 2.9), por lo que la proyección se dispone a lo largo de un meridiano central. Presenta la propiedad de conformidad, por lo que los ángulos entre líneas sobre el terreno se conservan correctamente. Este sistema de proyección se denomina universal porque divide el planeta en 60 zonas o **husos**, de 6^o de longitud, a partir de un meridiano de referencia. El huso 1 está centrado en 177^o O y se incrementa el número de huso hacia el este. El cilindro sobre el cual se

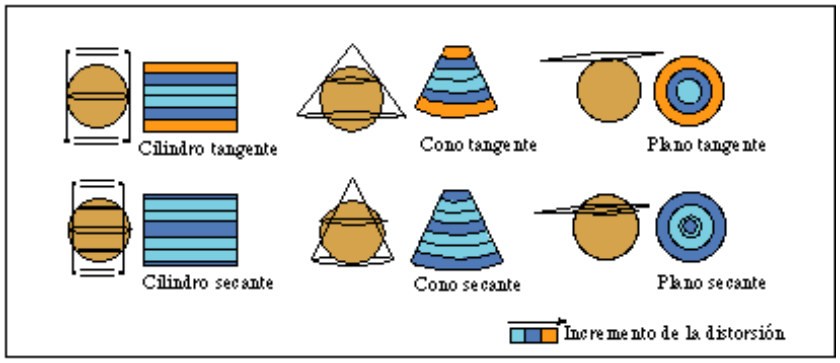


Figura 2.7: Tipos de proyección: cilíndrica, cónica y plana o acimutal (fuente: Escuela Universitaria de Ingeniería Técnica Topográfica de la Universidad Politécnica de Madrid).

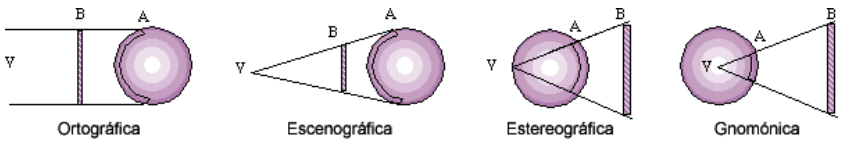


Figura 2.8: Tipos de proyección acimutal o plana (fuente: Escuela Universitaria de Ingeniería Técnica Topográfica de la Universidad Politécnica de Madrid).

Nombre	Conforme	Equivalente	Equidistante	Tipo
Mercator	•			Cilíndrica
Transverse Mercator	•			Cilíndrica transversa
Universal Transverse Mercator	•			Cilíndrica transversa
Gall-Peters		•		Cilíndrica
Plate Carée			•	Cilíndrica
Cassini			•	Cilíndrica transversa
Conforme de Lambert	•			Cónica
Cónica equidistante			•	Cónica
Albers		•		Cónica
Bonne		•		Pseudocónica
Estereográfica	•			Acimutal
Equivalente de Lambert		•		Acimutal
Equidistante de Postel			•	Acimutal
Gnomónica				Acimutal
Ortográfica				Acimutal

Cuadro 2.4: Principales proyecciones cartográficas y sus características.

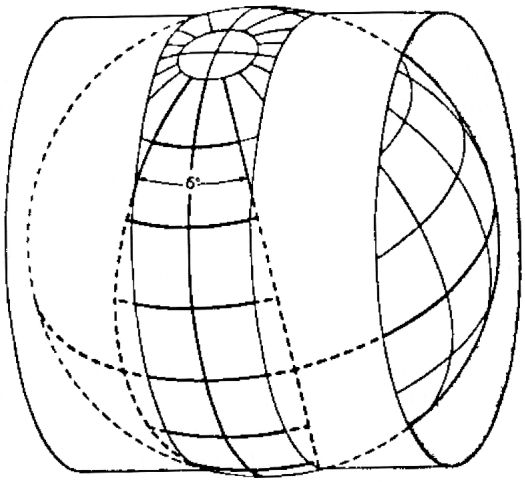


Figura 2.9: Proyección UTM.

proyecta es secante y no tangente a la superficie de la Tierra, por lo que existen dos líneas alrededor de la Tierra en cada huso donde no existe distorsión. Esto quiere decir que en el centro del huso sí se produce distorsión, pero en contrapartida se minimiza la distorsión en los extremos, lo cual sirve para reducir la distorsión media en toda la zona ($\pm 1/2.500$ máximo). Los husos se extienden desde 80^o S hasta 84^o N. El sistema UTM define también unas divisiones en latitud que forman zonas de 8^o desde el ecuador hacia los polos y que tienen asignada una letra. En la figura 2.10 se ilustra esta división en zonas de 6^ox8^o. Las coordenadas manejadas en UTM están representadas en metros y son siempre positivas, representando independientemente la mitad correspondiente al hemisferio norte y la correspondiente al hemisferio sur. Para asegurarse de que las coordenadas de toda la zona cubierta son positivas, en longitud se toma como origen el meridiano central aplicando un desplazamiento (*false easting*) de 500 km y en latitud se toma como origen el ecuador, aplicando un desplazamiento (*false northing*) de 10.000 km en el caso del hemisferio sur. Por lo tanto, una referencia geográfica en UTM incluirá el huso o zona en el que está representado, el hemisferio (N o S) y las dos coordenadas que en inglés se suelen denominar *easting* y *northing*.

A pesar de estar muy extendido su uso, la proyección Mercator tiene una importante desventaja, que es la distorsión de las áreas y distancias. En 1974, el Dr. Arno Peters, historiador y cartógrafo, presentó en Alemania la proyección que lleva su nombre. Se trata de una proyección cilíndrica equivalente, y por lo tanto preserva las áreas relativas de las diferentes zonas, a costa de sacrificar la forma (no se conservan los ángulos correctamente). La proyección de Peters, representada en la figura 2.11, ha suscitado bastante polémica, creando a la vez numerosos defensores y detractores.

Sus defensores esgrimen el argumento de que las proyecciones como Mer-

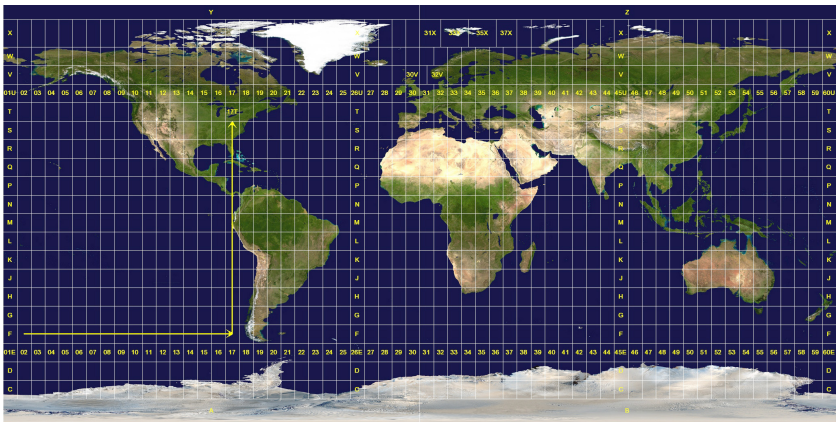


Figura 2.10: División de zonas de la proyección UTM.

cator fueron creadas en la época en que los navegantes surcaban los océanos en barcos de madera, impulsados por el viento y orientándose por las estrellas. Resultaba de utilidad para la navegación porque las líneas rectas en esta proyección poseen un rumbo constante. Y por este motivo todavía permanece en uso para la navegación tanto marítima como aérea. Además, las formas en la proyección son similares a las reales, pero es precisamente este el motivo de que los tamaños relativos se distorsionen completamente.

Las distorsiones de tamaño se incrementan conforme nos alejamos del ecuador. Los cartógrafos llamaron a este efecto que se acentúa al aproximarnos a los polos como "el problema de Groenlandia". En una proyección Mercator, Groenlandia aparece con un tamaño similar al de África, cuando en la realidad, el área de África es 14 veces superior al de Groenlandia.

Muchos de los argumentos esgrimidos a favor de la proyección de Peters frente a la Mercator son políticos y psicológicos. Se dice que la proyección Mercator era adecuada en la época colonial donde las principales potencias estaban situadas en Europa. Este contiente ocupa el lugar central a un tamaño muy superior al real en relación a los otros continentes. Su uso en la actualidad es en gran parte debido a la inercia.

La proyección de Peters no es del todo original, lo cual fomentó notablemente la polémica entre los cartógrafos. Está basada en la denominada proyección "ortogonal" de Gall, publicada en la *Scottish Geographical Magazine* en 1885 por el clérigo James Gall. Posteriormente, diversos cartógrafos realizaron variantes de esta proyección, pero fue Peters el que realmente le dio el alcance mediático que la hizo muy conocida, apoyado por los movimientos de justicia social en esa época. En la actualidad se conoce frecuentemente como proyección **Gall-Peters**, y es ampliamente utilizada por las organizaciones humanitarias y de ayuda a otros países, puesto que representa los países subdesarrollados con sus proporciones reales. Se podría decir que es una proyección políticamente correcta según cierto punto de vista, frente a

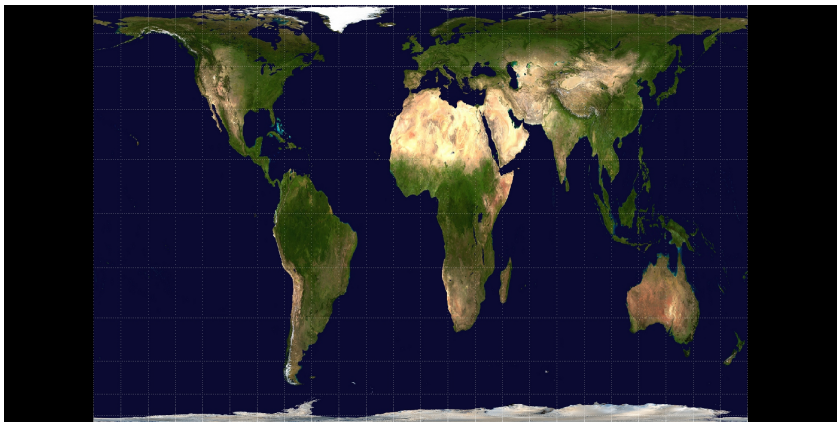


Figura 2.11: Proyección Gall-Peters.

la denominada por Peters y sus seguidores como "cartografía imperialista"[6] que representa los países desarrollados en el centro y con un tamaño muy superior al real, mientras que África y Sudamérica tienen un tamaño proporcionalmente mucho menor.

En el proyecto SANTI, mencionando en el capítulo de introducción, se trabajó para todos los datos georreferenciados con la proyección UTM correspondiente al huso 29N. Esta proyección resultó adecuada porque la zona representada no se extendía más allá de dicho huso.

En la actualidad se utilizan numerosos sistemas de proyección, según la zona que interesa representar y según las características que se desean conservar sin distorsión. Algunos de los principales sistemas se muestran en la tabla 2.4 y se ilustran en las figuras 2.12 y 2.13. Para más información existe una amplia bibliografía al respecto [46, 142, 141].

Para realizar la conversión entre diferentes sistemas de proyección existen paquetes de *software* como PROJ [23]. Este *software*, disponible bajo licencia MIT, ha sido desarrollado por el U.S. Geological Survey (USGS) y está basado en gran parte en el *General Cartographic Transformation Package* (GCTP), desarrollado anteriormente por la misma entidad. Se trata de un *software* muy interesante no sólo por la cantidad de tipos de proyección que tiene implementados (más de 70 en la versión actual), sino porque además al disponer del código fuente es fácil integrar su funcionalidad en otros proyectos. También es fácil su integración dentro de *scripts*, puesto que su interfaz en línea de órdenes está diseñada para usarse como un filtro de UNIX, es decir, se utiliza la entrada estándar (*stdin*) para enviarle las coordenadas a convertir y la salida estándar (*stdout*) para obtener el resultado.

[6] De hecho Peters hace referencia en su obra a la "evil Mercator".

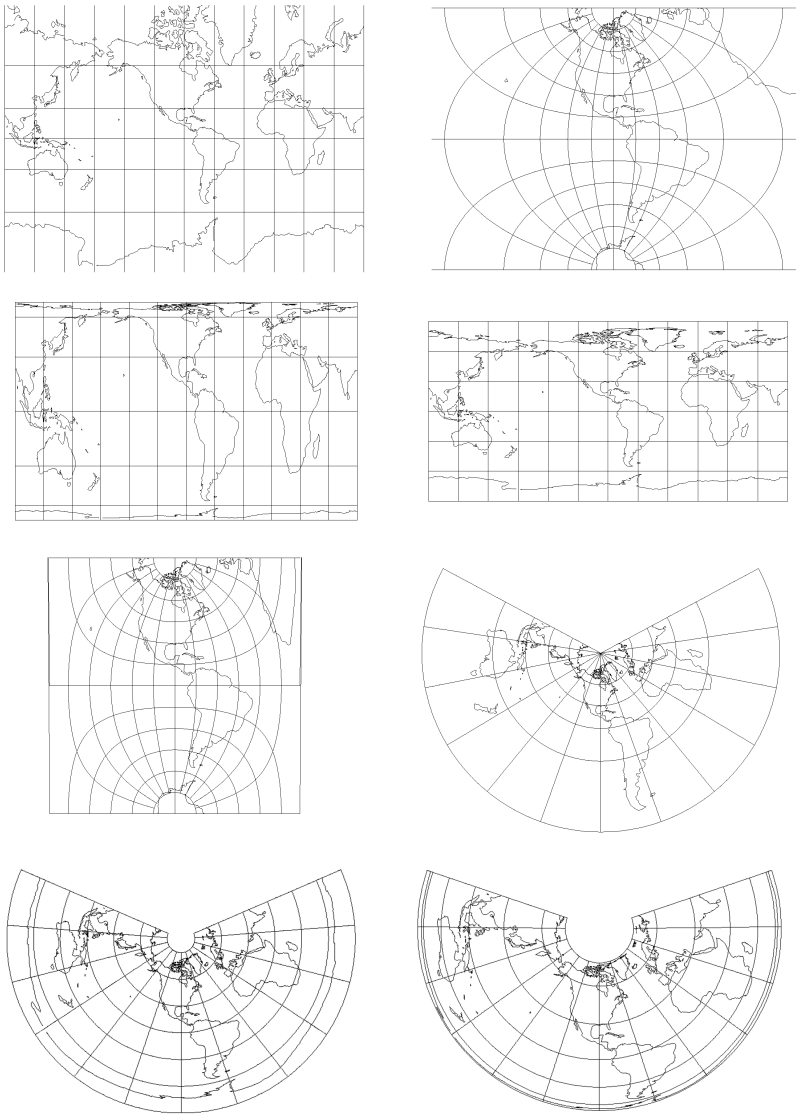


Figura 2.12: Proyecciones cartográficas. De izquierda a derecha y arriba a abajo: Mercator, Mercator transversa, Gall-Peters, Plate Carée, Cassini, Conforme de Lambert, Cónica equidistante y Albers (fuente: USGS [69]).

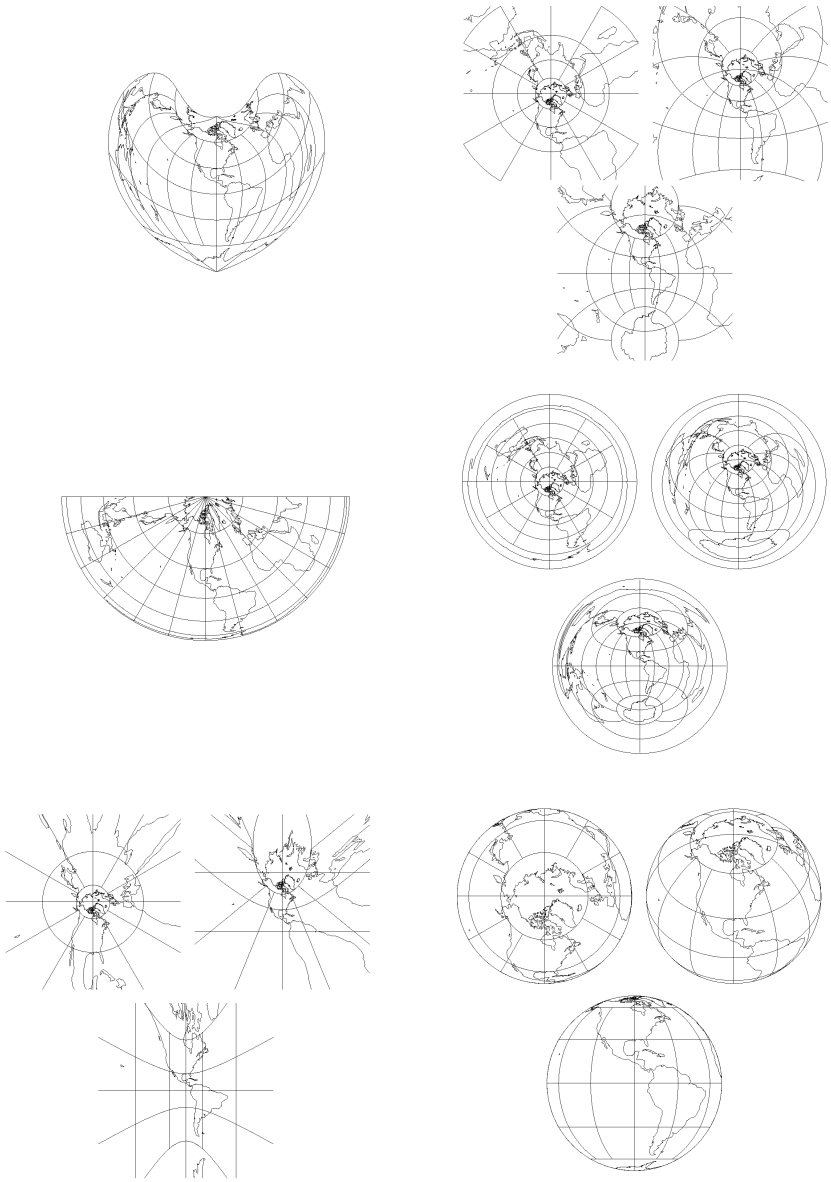


Figura 2.13: Proyecciones cartográficas. De izquierda a derecha y arriba a abajo: Bonne, Estereográfica, Equivalente de Lambert, Postel, Gnomónica y Ortográfica (fuente: USGS [69]).

2.2.4. *Software* SIG

Las operaciones que debe poder realizar un SIG incluyen la integración, el almacenamiento, la edición, el análisis, la compartición y la visualización de información geográfica. El presente trabajo resulta afectado en mayor o menor medida por casi todas ellas, pero principalmente nos centraremos en la última: la **visualización** de la información geográfica.

Trataremos el **almacenamiento** de dicha información con el objetivo de estudiar cuál es la organización o la estructura que posibilita un acceso a la misma lo más eficiente posible. Por este mismo motivo se estudian los aspectos de **transmisión** de esa información y el uso de cachés a diferentes niveles para mejorar la velocidad de transferencia y reducir el tiempo de acceso a los datos.

Otras tareas como la **edición** y el **análisis** se delegan en *software* de SIG externo, diseñado expresamente para ese tipo de trabajo. Únicamente se plantea una cierta posibilidad de interacción sobre el modelo 3D para seleccionar información procedente del SIG y realizar tareas relativamente sencillas con ella, proponiendo una comunicación bidireccional y continua con el SIG para la consecución de las tareas más complejas de análisis, edición, creación y mantenimiento de los datos.

En la actualidad existen numerosos SIG en funcionamiento. En algunos casos se trata de *software* desarrollado a medida por instituciones para su uso interno, principalmente militares y gubernamentales. En otros casos es *software* comercial disponible en el mercado con precios y prestaciones muy variados, y por último hay numerosos proyectos abiertos y disponibles como *software* libre.

Software SIG comercial

En el caso de los SIG comerciales, ESRI [7] (*Environmental Systems Research Institute, Inc.*) es la empresa lider en el mercado, con su producto estrella **ArcGIS**. ArcGIS es un conjunto de herramientas integradas que permiten crear, visualizar, analizar, cartografiar y servir datos geográficos de forma local o a través de redes de comunicaciones, incluida por supuesto Internet. Una de las herramientas más interesantes de este paquete es **ArcGIS 3D Analyst**, que incluye un visualizador **ArcGlobe**, que permite examinar la información geográfica sobre el globo terrestre en lugar de hacerlo sobre un plano bidimensional. Recientemente, ESRI ha anunciado otra herramienta de capacidades similares que ofrecerá de forma gratuita: **ArcGIS Explorer**.

Otra de las compañías más fuertes en *software* geoespacial era ERDAS, recientemente adquirida por Leica Geosystems [14]. Su producto *software* más importante en el campo de los SIG es **ERDAS IMAGINE**. Al igual que ArcGIS, se trata de un paquete de aplicaciones integradas, en este caso más orientadas hacia el tratamiento de imágenes geográficas. Su producto

IMAGINE VirtualGIS es un *software* de análisis visual que provee las funcionalidades de un SIG y la capacidad de representar la información en 3D. En este sentido es similar al ArcGlobe o ArcGIS Explorer de ESRI. ERDAS/Leica Geosystems también dispone de *plug-ins* para ArcGIS, como el **Image Analysis**.

La compañía InterGraph [12] posee su propia colección de herramientas SIG, llamada **GeoMedia**. Para visualización de cartografía a través de *web* disponen del producto **OGC Viewer**, y para visualizar el terreno en 3D de forma interactiva, el *software* de Geomedia se integra con el **TerraExplorer Pro** de Skyline [26].

IDRISI es un *software* comercializado por los laboratorios Clark Labs [5], perteneciente a la Clark University. Está compuesto por pequeños módulos (más de 250 en la versión actual) para realizar diferentes tareas de análisis, tratamiento y visualización de la información geoespacial. La herramienta de visualización interactiva de terreno en 3D se llama **Fly Through**.

En el mundo de la ingeniería civil está muy extendido el uso de los productos **Microstation** de Bentley [3] y **AutoCAD Map 3D** de Autodesk [1], que combinan características de *software* de diseño asistido por ordenador (CAD) y SIG.

Existen otras compañías, como ER Mapper [6] con productos relacionados con los SIG pero dedicados al procesamiento de imágenes geográficas exclusivamente y pensadas para usar como complemento a *software* de SIG como los anteriormente mencionados.

Blue Marble Geographics [4], por su parte, ofrece *software* para la conversión de datos geográficos: conversión de coordenadas, reproyección de imágenes, conversión de formatos vectoriales.

Para la generación de cartografía a partir de datos SIG, la compañía Avenza [2] dispone de productos como **Mapublisher** y **Geographic Imager**, que permiten realizar estas tareas desde Adobe Publisher o Macromedia Freehand y Adobe Photoshop respectivamente.

Software SIG libre

Respecto al *software* libre, existen diversas herramientas, muchas de las cuales están recopiladas dentro de una fundación llamada **OSGeo** [16]. La *Open Source Geospatial Foundation* (OSGeo) es una organización sin ánimo de lucro que nació con el objetivo de promover y dar soporte al desarrollo de tecnología y datos geoespaciales abiertos. Da soporte legal, organizativo y financiero a la comunidad de código abierto geoespacial. Supone una entidad legal independiente a la cual los miembros de la comunidad pueden contribuir con código, fondos u otros recursos, con la seguridad de que serán mantenidos y empleados para el beneficio público.

Estas herramientas abiertas incluyen aplicaciones SIG típicas para análisis, tratamiento y visualización de datos geoespaciales, como **GRASS** [9]

(*Geographic Resources Analysis Support System*), **OSSIM** [20] (*Open Source Software Image Map*), **Quantum GIS** [24], **JUMP** [27] (*JUMP Unified Mapping Platform*) u **OpenJUMP** [17], basado en el anterior.

También existen otro tipo de aplicaciones de código abierto que permiten publicar información geográfica a través de la *web* o integrar servicios cartográficos en páginas *web* obteniendo estos datos de diferentes servidores remotos. Ejemplos de este tipo de aplicaciones disponibles como *software* libre son **MapServer**, **MapBuilder**, **OpenLayers**, **Mapbender** o **MapGuide**, todas ellas auspiciadas por OSGeo.

Para el desarrollo de aplicaciones relacionadas con los SIG, existen algunas librerías de código abierto para el manejo de datos geoespaciales como **GeoTools** [11], **PROJ** [23] o **GDAL/OGR** [8] (*Geospatial Data Abstraction Library*), esta última ampliamente utilizada en el presente trabajo para los preprocesos de la información cartográfica.

Por último, existe también *software* libre para la gestión de bases de datos geoespaciales. El más conocido está implementado como una extensión para uno de los mejores sistemas de gestión de bases de datos (SGBD) disponibles en código abierto: PostgreSQL [22]. Esta extensión, que añade soporte para datos geográficos en las bases de datos, se denomina **PostGIS** [21]. Está desarrollada por Refractions Research y permite servir esos datos a aplicaciones de SIG. PostGIS posee el certificado[7] del OGC [15].

Como ejemplo de combinación de diferentes proyectos de *software* SIG libre, existe un sistema denominado **Kosmo**, que integra parte del *software* mencionado previamente. Está desarrollado por SAIG [62], una empresa dedicada a los servicios basados en *software* SIG libre. Kosmo está basado principalmente en JUMP y GeoTools, y su principal enfoque es la interfaz de usuario. El objetivo de SAIG es la creación de un entorno SIG corporativo flexible y potente basado en *software* libre. En definición de sus propios autores se trata de "*una plataforma SIG libre corporativa como alternativa real a los sistemas comerciales*".

Iniciativas similares en cierto modo a la anterior están siendo desarrolladas por diferentes administraciones públicas. Algunos ejemplos son **gvSIG**[63] y **GeoPISTA**[10].

gvSIG está promovido por la Generalitat Valenciana a través de la Consellería de Infraestructuras y Transporte, supervisado por la Universidad Jaume I y desarrollado principalmente por la empresa IVER Tecnologías de la Información. Está desarrollado en Java y publicado bajo licencia GNU GPL.

GeoPISTA está planteado como un SIG para los ayuntamientos y en él participan numerosas entidades administrativas, tanto estatales como regionales y locales (Ministerios, Comunidades Autónomas, Diputaciones Provinciales y Ayuntamientos). Se define como un "*Sistema de Información Ter-*

[7] *OGC Compliant* con el perfil "tipos y funciones".

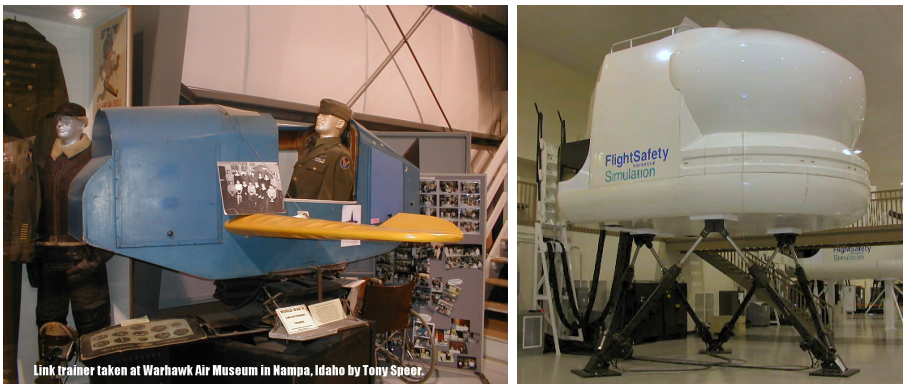


Figura 2.14: Simuladores de vuelo. Link Trainer (izquierda) es uno de los sistemas pioneros, anterior a la simulación visual utilizando computadores digitales. A la derecha un simulador actual con movimiento de alta precisión y 6 grados de libertad (©2005 FlightSafety International).

ritorial para Ayuntamientos que, partiendo de la base de una cartografía, pueda georreferenciar tanto la información como la propia gestión municipal, proporcionando así a los Ayuntamientos y a los ciudadanos servicios en línea basados en sistemas de información geográfica (SIG)".

Es interesante destacar que así como en el *software* comercial existen diversas soluciones que con mayor o menor calidad y prestaciones proporcionan una vista 3D interactiva de la información geográfica, en el *software* libre, este tipo de sistemas es difícil de encontrar.

2.3. Visualización de terreno

Aunque las aplicaciones de la visualización en tiempo real de terreno son muy diversas en la actualidad, sus orígenes están ligados al campo de la simulación de vuelo. De hecho, es muy probable que la primera de las aplicaciones de la visualización interactiva de terreno tridimensional fueran los simuladores de vuelo para el entrenamiento de pilotos, tanto en la industria aeronáutica como en el ámbito militar. Por razones evidentes, tanto económicas como de seguridad, el entrenamiento de los pilotos no se debe realizar directamente en un vuelo real. Por este motivo, surgen los simuladores de vuelo a principios del siglo XX. Uno de los primeros es el **Link Trainer** (figura 2.14), desarrollado durante la Primera Guerra Mundial.

Inicialmente, los simuladores de vuelo se limitaban a reproducir el movimiento de la nave. Posteriormente se añadieron controles para que el piloto pudiese interactuar con ella. Sin embargo, en estos primeros sistemas no se disponía de pantalla y por tanto no había una **simulación visual**.

Los primeros simuladores con información visual y movimiento completo surgieron a finales de la década de los 50, utilizando maquetas del terreno y

moviendo una cámara sobre dichas maquetas, cuya imagen se enviaba a un monitor de TV en la cabina del piloto.

Fue en los años 60 cuando se comenzaron a utilizar computadoras digitales para la visualización del terreno en los simuladores de vuelo, y en las siguientes décadas se mejoró notablemente la tecnología de las pantallas para conseguir un alto grado de realismo y sensación de inmersión.

La visualización del terreno en los simuladores de vuelo tiene unas características radicalmente distintas a los SIG, mencionados en la sección 2.2. Si en el caso de estos últimos lo más importante es la precisión de los datos o la calidad de la información presentada, en un simulador de vuelo, el tiempo de respuesta del sistema es de vital importancia. Por esto se habla de sistemas o simuladores en **tiempo real**, entendiendo como tales aquellos en que se debe producir la respuesta a las entradas en un tiempo no superior a un límite preestablecido.

El pilotaje de una aeronave es una actividad altamente dinámica. Las condiciones del entorno pueden variar instantáneamente y las respuestas deben ser suficientemente rápidas. Cuando se realiza la simulación, el piloto debe estar en unas condiciones lo más cercanas posible a las reales. Si la visión del entorno tiene un retardo apreciable respecto a lo que sucedería en la realidad, o no se actualiza lo suficientemente rápido como para dar la sensación de movimiento suave, el entrenamiento no será efectivo. Esto es especialmente crítico en el entrenamiento de pilotos militares, puesto que las condiciones son extremas. Un mínimo retardo en la reacción del piloto puede suponer la diferencia entre la vida y la muerte.

En un sistema de visualización en tiempo real trabajamos con dos conceptos distintos aunque relacionados, y que a veces pueden dar lugar a confusión: la latencia (o tiempo de respuesta) y la frecuencia de actualización (o frecuencia de refresco).

Por un lado hablamos de la **latencia** del sistema refiriéndonos al tiempo que transcurre entre que el usuario genera una entrada y se produce una salida como respuesta a esa entrada. Por ejemplo la entrada puede consistir en accionar la palanca de dirección en la cabina del avión y la salida será la imagen vista en la cabina desde la posición del avión que resulta del movimiento producido por ese movimiento de la palanca.

La **frecuencia de actualización** de la pantalla, por otra parte, es el número de veces por segundo que se obtiene una nueva imagen de la vista del terreno desde la cabina del avión. Si esta frecuencia no es lo suficientemente alta, se producirá la sensación de estar viendo una serie de imágenes estáticas con saltos de una a otra en lugar de percibir un movimiento continuo y suave como el que se vería en la realidad.

Como ya se ha explicado, ambos aspectos, latencia y frecuencia de refresco, son críticos en un simulador. Por este motivo, en los primeros simuladores, dadas las limitaciones de la tecnología del momento, se sacrificaba la calidad de la visualización del terreno en favor de la velocidad de respues-

ta del sistema. El terreno se utilizaba principalmente como soporte visual para contribuir a percibir el movimiento respecto a una referencia fija. Por este motivo, no interesaba que los datos se correspondiesen con la realidad existente en el terreno, sino simplemente que pareciese lo más real posible dentro de las posibilidades técnicas del momento.

Los primeros simuladores visuales utilizaban los llamados generadores de imagen o IG (*Image Generators*), computadoras con *hardware* gráfico especializado y de elevado coste. Estos IG estaban desarrollados por compañías como Evans & Sutherland, Silicon Graphics, CAE o MacDonell Douglas. En la actualidad, los ordenadores personales han entrado en el mercado de la simulación [105, 78], en forma de baterías o *clusters* de PCs con potentes GPUs. Algunas compañías como Quantum3D [25] u ORAD [19] están especializadas en este tipo de sistemas.

Los requerimientos mínimos para una aplicación de simulación visual se suelen situar alrededor de los 60 fotogramas por segundo (fps) para la frecuencia de actualización. Respecto a la latencia, existen diversos estudios sobre el tema que hablan de tiempos de hasta 200 ms [56]. Este tiempo de respuesta es muy variable según la nave a pilotar. Un avión comercial con capacidad para un gran número de pasajeros no es tan ágil como un caza, y por lo tanto los requisitos mínimos en cuanto al tiempo de respuesta de la simulación serán diferentes. Buscando un límite a la capacidad de la percepción humana en este aspecto, podemos mencionar el trabajo de Robinson y Mania [125] quienes determinan que los pilotos experimentados pueden apreciar latencias a partir de los 15 ms.

Además de la simulación de vuelo, en las últimas décadas han surgido otros muchos campos de aplicación para la visualización de terreno. Algunos de los más importantes se enumeran a continuación.

- Simulación. Planificación y ensayo de operaciones.
- Entrenamiento.
- Planificación y gestión de recursos (agricultura, meteorología, hidrología, climatología, minería, ingeniería forestal, etc).
- Planificación y gestión de infraestructuras (telecomunicaciones, catastro, urbanismo, ingeniería civil, etc).
- Turismo. Promoción turística, turismo virtual, planificación de viajes.
- Educación. Apoyo a asignaturas relativas a la geografía, ciencias sociales, ciencias naturales, etc.
- Entretenimiento. Videojuegos.

Además de las aplicaciones mencionadas, existen muchas otras más específicas o menos habituales. Algunos ejemplos de ellas se mencionan en un

discurso [80] pronunciado en enero de 1998 por el entonces vicepresidente de los EEUU Al Gore, como la diplomacia, la lucha contra el crimen, la preservación de la biodiversidad, la predicción del cambio climático o el incremento de la productividad en la agricultura.

La lista sería muy larga, pero no todas las aplicaciones son tan exigentes en cuanto a la respuesta del sistema como los simuladores de vuelo. De las mencionadas en el listado anterior, tal vez las que tienen requisitos más estrictos en este aspecto son las de simulación, entrenamiento y entretenimiento. De hecho, cada vez está más difusa la línea que separa las aplicaciones de entrenamiento militar y planificación de misiones de los videojuegos bélicos o de acción [30]. Sin embargo, dada la vertiginosa evolución del *hardware* gráfico de consumo, los usuarios son cada vez más exigentes y demandan una gran calidad de visualización al tiempo que esperan una respuesta inmediata del sistema en todo tipo de aplicaciones que visualizan el terreno en 3D.

En cualquier caso, lo que es indiscutible es la convergencia sufrida entre los campos de la visualización en tiempo real de terreno 3D y los SIG, hasta el punto de que las diferencias cada vez están menos claras para el usuario neófito, que puede llegar a confundirlos.

Se cual sea la aplicación que se le de, la visualización de terreno 3D en tiempo real se basa principalmente en un modelo geométrico de elevación de la superficie del terreno, sobre la cual se suele aplicar una o más texturas. Estas texturas consisten habitualmente en imágenes de satélite o fotografía aérea, aunque podría tratarse de un mapa de carreteras o cualquier otro mapa temático con información *raster* 2D asociada al terreno.

En las próximas secciones se tratarán diversos aspectos de la visualización de terreno en tiempo real, incluyendo geometría y textura. Se analizan los trabajos previos y el estado del arte de este campo. Finalmente se profundizará en la combinación de información procedente de SIG con el terreno tridimensional, objetivo de esta tesis doctoral.

2.4. Modelos topográficos

Los sistemas de visualización en tiempo real de geometría del terreno se pueden clasificar en función de diversos criterios. En primer lugar las estructuras geométricas en que se basan para definir el modelo de la topografía del terreno.

Una de las formas habituales en cartografía para representar la elevación del terreno son las curvas de nivel. Una curva de nivel une puntos de la superficie del terreno que tienen la misma altura. Se corresponde con el contorno de una sección horizontal de dicho terreno. Estas curvas son equidistantes respecto al parámetro que representan, en este caso la altitud.

Aunque esta representación en forma de curvas de nivel puede resultar de utilidad en un mapa en papel, en el campo de los gráficos por computador

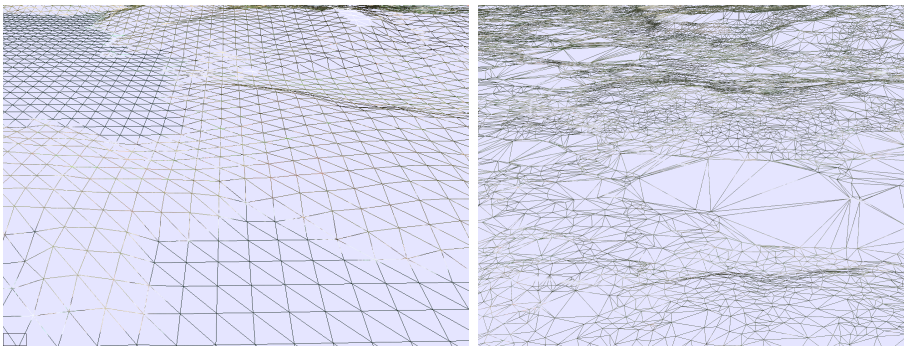


Figura 2.15: Ejemplos de mallas regular e irregular (TIN).

no se suele utilizar debido a su dificultad de manejo. En su lugar, se suelen utilizar aproximaciones basadas en mallas de polígonos, que se pueden clasificar en dos tipos: mallas regulares (en inglés *grids*) o mallas irregulares de triángulos (en inglés *triangulated irregular networks*) que habitualmente se denominan TIN. En la figura 2.15 se muestran ejemplos de ambos tipos de malla, descritos en los siguientes apartados.

Cada sistema de modelización del terreno tiene sus ventajas y sus inconvenientes, de forma que según la aplicación específica que se le vaya a dar y los objetivos planteados, se elegirá una u otra. Por este motivo es muy importante conservar la independencia entre el sistema de gestión de geometría y de textura del terreno, uno de los objetivos primordiales de este trabajo. El sistema de texturizado de imagen *raster* y datos vectoriales procedentes de SIG propuesto en esta tesis funcionará independientemente del motor de geometría que se utilice, de las técnicas o algoritmos en los que se base y de la forma de modelizar el relieve del terreno.

2.4.1. Mallas regulares (*grid*)

Las mallas regulares definen una matriz de muestras de altura equiespaciadas en ambos ejes del plano horizontal. Presentan numerosas ventajas, principalmente por su sencillez de manejo en diferentes aspectos:

Render La topología regular facilita la generación de triángulos para *renderizar* en el *hardware* gráfico. Además, estos triángulos se pueden agrupar de forma inmediata en estructuras muy eficientes para la GPU, como tiras (*tristrips*) o abanicos (*trifans*) de triángulos.

Niveles de detalle Es habitual el uso de mallas regulares, especialmente en tamaños potencia de dos, para facilitar la reducción del detalle eliminando muestras intermedias. De la misma forma, se puede aumentar el detalle generando vértices intermedios mediante diversas técnicas de

interpolación e introducción de detalles de alta frecuencia a partir de funciones de ruido [107].

Cálculos sobre el terreno La topología regular de estas mallas facilita también la realización muy eficiente de cálculos como pueden ser las intersecciones con el terreno.

Almacenamiento En las mallas regulares sólo se necesitan almacenar los valores de elevación en cada punto de la malla, puesto que las coordenadas en el plano horizontal están implíctias. De esta forma, además de reducir el espacio necesario para los vértices a un tercio aproximadamente y de no necesitar información acerca de la conectividad de dichos vértices, se facilita también el uso de formatos *raster* para los datos de elevación del terreno al igual que para las texturas.

La desventaja de las mallas regulares es que el detalle disponible del terreno es homogéneo en toda su extensión, de forma que en las zonas llanas posiblemente habrá información innecesaria al mismo tiempo que en zonas más escarpadas podría ser insuficiente. Además, como la toma de muestras sigue un patron fijo, los cambios bruscos de pendiente no se pueden representar de forma adecuada. Esto se puede apreciar fácilmente en zonas como acantilados, gargantas y cañones (ver figura 2.16). Para representar de forma adecuada estas situaciones es necesario poder tomar muestras cercanas en las zonas de cambio de pendiente.

2.4.2. Mallas irregulares (TIN)

Las mallas o redes irregulares de triángulos (TIN) se forman a partir de una colección de vértices correspondientes a cotas del terreno que no siguen una organización fija como en el caso de las mallas regulares anteriormente descritas.

La creación de los triángulos que forman la TIN se suele realizar de forma que cada triángulo sea lo más cercano posible a un equilátero, evitando triángulos largos y estrechos. Este proceso de teselado produce como resultado una lista de vértices, otra lista de aristas que unen pares de los vértices anteriores y una lista de triángulos formados por tres aristas no colineales de las anteriores.

Una de las formas habituales de realizar este proceso es mediante la denominada triangulación de Delaunay. Esto divide la zona ocupada por los vértices en triángulos de Delaunay que se caracterizan por la condición de que la circunferencia circunscrita de dicho triángulo no debe contener vértices de ningún otro triángulo de la malla. Esta triangulación está directamente relacionada con la división del espacio mencionado en los llamados polígonos o celdas de Voronoi, Thiessen o Dirichlet. Estos polígonos, asociados cada uno a un vértice de la malla, cumplen la condición de contener todos los

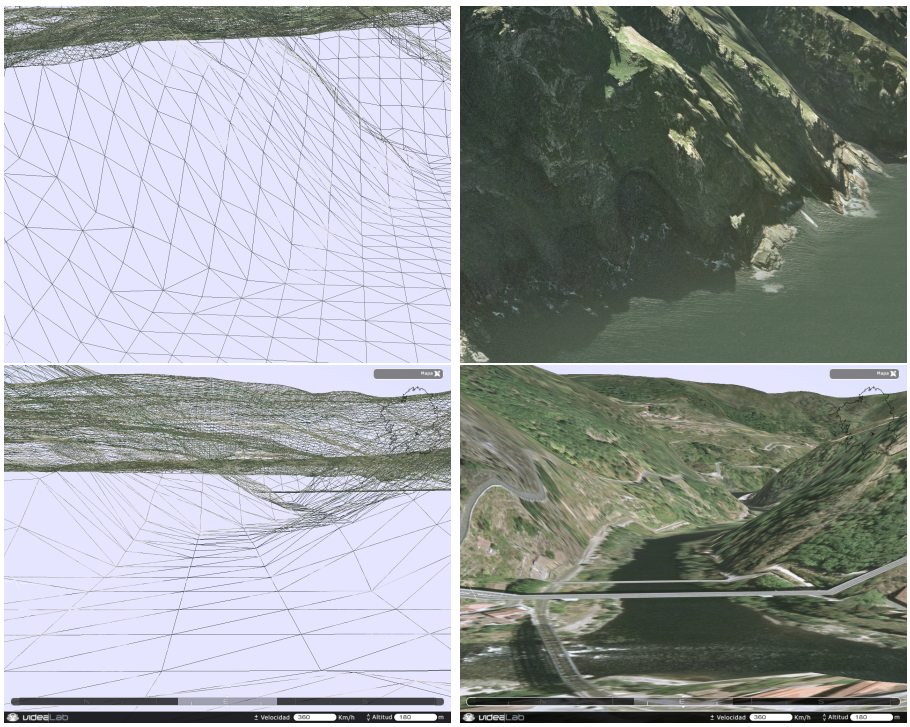


Figura 2.16: Problemas de las mallas regulares en zonas con cambios bruscos de pendiente.

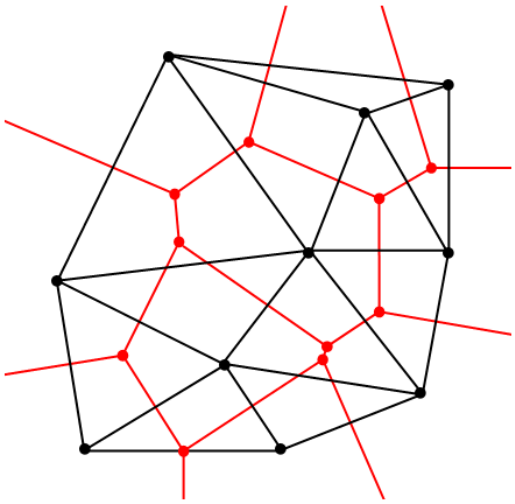


Figura 2.17: Triangulación de Delaunay (negro) y celdas de Voronoi (rojo).

puntos del espacio cuyo punto más cercano es el correspondiente en la malla a dicho polígono y son muy utilizados para el cálculo de áreas de influencia. En la figura 2.17 se muestra el resultado de la triangulación de Delaunay (representada en negro) para un conjunto de puntos y las correspondientes celdas de Thiessen (representadas en rojo).

Dado que no hay una estructura regular, es necesario indicar las aristas de los triángulos que forman la malla además de la posición en tres dimensiones de los vértices. Por lo tanto, en cuanto a necesidades de almacenamiento del modelo topográfico del terreno, las TIN tienen serias desventajas frente a las mallas regulares, que sólo necesitaban almacenar las coordenadas en el plano de una esquina de la zona modelizada, la separación entre cotas y un valor de elevación para cada cota.

A la hora de dibujar la geometría (*render*), también resulta más costoso en tiempo de cálculo, puesto que no es tan sencillo agrupar los triángulos en tiras o abanicos, y se suelen enviar al *pipeline* gráfico como triángulos sueltos.

Desde el punto de vista del texturizado, el uso de TIN dificulta la división del terreno en rectángulos a los cuales aplicar una textura correspondiente a esa zona. Esto es un problema grave en algunos sistemas de texturizado, aunque en la técnica propuesta en este trabajo está solucionado, tal y como se describirá más adelante.

La gran ventaja de las TIN es su facilidad para adaptarse a las características del terreno, de forma que se tomen más cotas en las zonas más escarpadas y en las posiciones más adecuadas para reproducir la orografía del terreno y menos cotas en las zonas llanas donde esa información sería redundante o innecesaria. Las TIN solucionan los problemas de las mallas

regulares en zonas como valles abruptos y acantilados, comentados anteriormente e ilustrados en la figura 2.16.

Además de las zonas del terreno que hacen aconsejable el uso de TIN (o bien un aumento de la densidad de la malla regular, lo cual puede resultar prohibitivo para el sistema gráfico) debido a las características del propio terreno, existen situaciones en las que se hace necesario el uso de TIN por otros motivos. Un ejemplo de este tipo de situaciones es la integración dentro del terreno de modelos 3D de infraestructuras como pueden ser carreteras. En la figura 2.18 se puede apreciar un ejemplo donde el modelo de la carretera incluye los correspondientes taludes que encajan exactamente con el modelo digital del terreno. Este modelo del terreno ha sido adaptado al modelo 3D de la carretera desplazando los vértices de la TIN, proceso que no habría sido posible utilizando mallas regulares.

2.5. Visualización en tiempo real de terreno

El principal problema de la visualización de terreno en tiempo real es el elevado volumen de información que debe manejar el sistema gráfico. Esto afecta a dos aspectos: la memoria ocupada tanto por la geometría como por las texturas, y el coste en tiempo de cálculo de las tareas realizadas por el *rendering pipeline*[74][34].

Estas tareas consisten primeramente en procesar los vértices de la geometría 3D transformándolas a espacio pantalla según la configuración de la cámara. Posteriormente se procesan las primitivas formadas por esos vértices (ya transformados a espacio pantalla) en *fragments*[8] que se procesarán para realizar los cálculos de iluminación, mapeado de texturas, interpolación de color, fundidos y finalmente pasarán las pruebas finales (test de profundidad (algoritmo *Z-buffer*), test de canal alfa, test de estarcido (*stencil*), test de recorte (*scissor*)) que determinarán si finalmente el *fragment* en cuestión llega a convertirse en un pixel en el *frame buffer* o será descartado por no superar cualquiera de dichas pruebas. Este aspecto es absolutamente crítico, puesto que se debe generar cada fotograma en unos pocos milisegundos[9].

La carga de geometría viene determinada por dos factores: el detalle del modelo geométrico y la extensión del terreno visualizado. Es habitual que la cantidad de geometría que se maneja, tanto por complejidad como por extensión, desborde la capacidad de la memoria del sistema gráfico e incluso la memoria principal del ordenador y por supuesto la capacidad de proceso de polígonos.

En el caso de las texturas, el uso de memoria es más crítico, puesto

[8] Un *fragment* corresponde a la información necesaria para calcular el valor de un pixel y realizar las pruebas que pueden descartarlo.

[9] En el caso habitual de un *frame rate* de 60 fps, se dispone de 16,666 ms para el cálculo de cada fotograma.

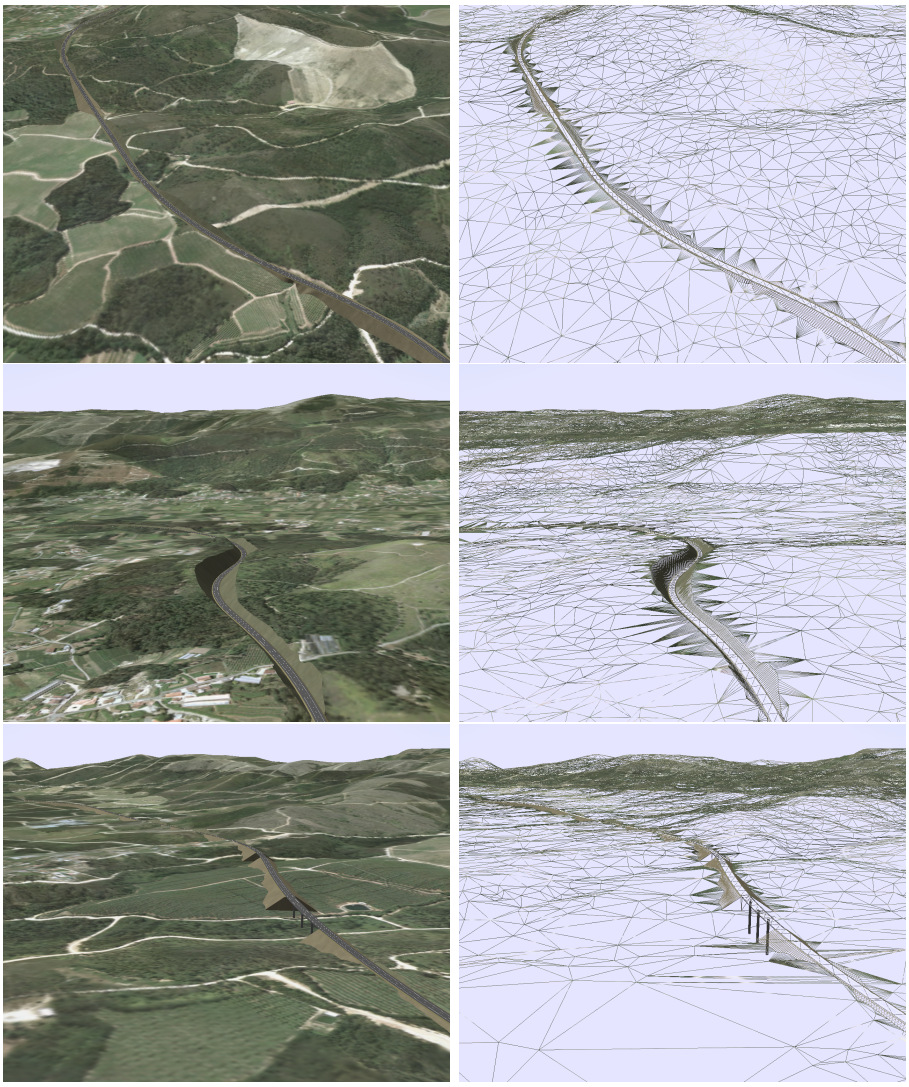


Figura 2.18: Uso de TIN para adaptar el modelo digital del terreno a las infraestructuras modeladas en 3D. (fuente: VideaLAB)

que habitualmente la cantidad de pixels que se utilizan puede llegar a ser varios órdenes de magnitud superior a la de vértices. Respecto al impacto en el tiempo de cálculo de la imagen, puesto que el proceso se realiza por *fragment*, no afecta el tamaño de la textura, sino la cantidad de pixels en pantalla cubiertos por los modelos texturizados y la complejidad en profundidad (*depth complexity*) de dichos modelos, es decir, el número de *fragments* que se procesarán por cada pixel. En el caso de la visualización de terreno, esto no suele ser especialmente crítico, puesto que con vistas en picado se llena la pantalla pero con una complejidad en profundidad[10] (*depth complexity*) cercana a uno, y en caso de vistas horizontales, se incrementará la profundidad de llenado en función de lo montañoso que sea el terreno y lo baja y horizontal que sea la línea de visión, sin embargo, la parte superior de la imagen (correspondiente al cielo) no contendrá terreno.

Sin embargo, hay una característica que posibilita el manejo eficiente del modelo del terreno sin desbordar los recursos del sistema. La cantidad de información que se visualiza en un momento dado está limitada por las características de la cámara y de la pantalla (o más concretamente de la ventana o *viewport*) en que se muestra. En concreto, está afectada por los ángulos del *frustum* de visión y por la resolución en pixels de la ventana de visualización. Cuando se visualiza un área del territorio muy extensa, no es necesario utilizar un detalle muy alto, puesto que no se llegaría a apreciar. De la misma manera, cuando realmente se aprecia un detalle muy fino, es porque se contempla un área reducida del territorio.

En el caso de la geometría, se puede considerar como umbral, respecto al tamaño de los polígonos, el tamaño de un pixel en pantalla. A partir de ese punto no se conseguirá ninguna mejora en el resultado de la visualización por incrementar el detalle de geometría, es decir, el número de polígonos.

Sin embargo, el problema no es tan sencillo como se acaba de plantear. En la visualización interactiva de terreno es habitual que la dirección de visión sea cercana a la horizontal. Esto provoca que realmente se necesiten diferentes detalles simultáneamente, más fino en las zonas próximas a la cámara y más burdo en las zonas lejanas.

En resumen, se puede decir que los dos principales problemas a resolver para la visualización de terreno en tiempo real son el desbordamiento de la memoria y el desbordamiento de la capacidad de proceso. En ambos casos existen técnicas que ayudan a resolver o paliar estos problemas. El desbordamiento de la memoria se soluciona a través de técnicas de **paginación** y el uso de **cachés**. Para evitar superar la capacidad de cálculo geométrico, se recurre a las técnicas de gestión de **niveles de detalle**.

[10] Denominamos complejidad en profundidad a la cantidad de *fragments* existentes en la posición de un pixel determinado o al promedio de este valor para todos los pixels de la imagen.

2.5.1. Paginación y cachés

La **paginación** consiste en dividir un gran volumen de información (que no cabe en el espacio de almacenamiento donde se va a utilizar) en fragmentos más pequeños, llamados genéricamente **páginas**. Este concepto surge en el campo de los sistemas operativos, donde se posibilita manejar un espacio de memoria superior a la memoria física disponible, denominado memoria virtual. El espacio de memoria virtual se divide en fragmentos de igual tamaño y se gestiona su transferencia entre el disco y el espacio disponible en memoria principal.

La paginación se utiliza en conjunto con el concepto de caché, que igualmente surge en el campo de los sistemas operativos. Una caché se puede definir como una memoria rápida que almacena una copia temporal de un subconjunto de cierta información cuyo acceso es más costoso en tiempo, ya sea porque está almacenada en un dispositivo de acceso más lento, o porque se trata de una información cuyo cálculo es costoso. El objetivo de la caché es almacenar aquella información que tiene probabilidades de ser solicitada próximamente y permitir en las siguientes ocasiones un acceso más rápido que si hubiese que acceder a los datos originales. De esta forma, el resultado final es una mejora en el tiempo de acceso.

Esto hace uso de algunas características o propiedades de **localidad** que se suelen dar en los patrones de acceso a los datos en la gran mayoría de aplicaciones informáticas. Existen diferentes tipos de localidad en los accesos a la información, pero los más habituales son la temporal y la espacial. Se habla de **localidad temporal** para hacer referencia al fenómeno que se produce cuando hay grandes probabilidades de que se soliciten los datos que han sido recientemente accedidos. Se habla de **localidad espacial** para hacer referencia al fenómeno que se produce cuando hay grandes probabilidades de que se soliciten los datos que están próximos[11] a los que se han sido recientemente accedidos.

Estas propiedades de localidad son especialmente válidas en el caso del manejo de información geográfica y la visualización de terreno, donde el usuario navega sobre el terreno siguiendo una trayectoria continua más habitualmente de lo que salta entre puntos lejanos. Por este motivo, el uso de cachés para los datos geográficos es fundamental y las ventajas son muy superiores a las obtenidas en otro tipo de aplicaciones.

Es habitual también el concepto de **jerarquía de cachés** o cachés en varios niveles. La situación habitual es una secuencia de dispositivos de almacenamiento en los que la velocidad va en decremento y la capacidad en aumento. Por ejemplo la caché de CPU en los ordenadores actuales no es única sino que existe una jerarquía de al menos un par de niveles (que gene-

[11] Esta proximidad se puede considerar física, por ejemplo los datos de los sectores contiguos en el disco, o lógica, por ejemplo en los elementos próximos de las estructuras de datos que se utilicen.

ralmente se denominan L1, L2 ...). A continuación de estas cachés suele haber otra caché de disco que almacena en memoria principal la información del disco que se accede frecuentemente, o que ha sido accedida recientemente, o que está próxima a la recientemente accedida. En algunos casos puede haber incluso más niveles, como una batería de cintas que almacenan información que debe ser conservada pero que rara vez será accedida. En este caso, los discos podrían contener una caché con parte de la información almacenada en cintas que ha sido recientemente solicitada.

2.5.2. Niveles de detalle

El objetivo de las técnicas de niveles de detalle es gestionar la cantidad de información que se presenta en cada zona de la imagen visualizada, de forma que no se produzca una carga de trabajo excesiva sobre el sistema gráfico en detalles que no serán apreciados por el espectador ni se omitan detalles allí donde sí hacen falta realmente.

La paternidad de las técnicas de niveles de detalle o LOD (por el inglés *levels of detail*) se atribuye a James Clark, que en 1976 publicó los principios básicos [51]. Según Clark, "el margen de complejidad de un entorno aumenta enormemente, mientras que la complejidad visible de una escena se mantiene dentro de unos límites fijos" y "el aumento de la complejidad de la escena, o el incremento de la información en la base de datos pierde valor cuando nos aproximamos a los límites de resolución de la pantalla". Clark expone de esta forma la redundancia en utilizar muchos polígonos o muchos texels para un elemento que sólo ocupa unos pocos pixels en pantalla.

En el trabajo de Clark, además de los niveles de detalle, se sugieren diversos conceptos fundamentales para el *render* en tiempo real, como el filtrado por visibilidad (*view-frustum culling*) o por oclusión (*oclussion culling*), la organización en estructuras jerárquicas denominadas grafos de escena (*scene graphs*) y su proceso en paralelo, y el mantener residente en la memoria de acceso rápido (en aquella época, la memoria principal del ordenador, hoy en día también habría que incluir la memoria de vídeo de la GPU) un subconjunto (que él denomina *working set*) de la escena total. Actualmente este tipo de técnicas se denominan en inglés como ***out-of-core rendering***. Este concepto está fuertemente relacionado con los de caché y paginación, anteriormente descritos.

En la actualidad existen muchos trabajos acerca de los niveles de detalle, especialmente centrados en mallas poligonales, aunque estas técnicas se pueden aplicar en muchas otras situaciones, como splines, voxels, superficies implícitas o imágenes [108]. Dentro de las técnicas de niveles de detalle se diferencian tres tipos de LOD: **discretos**, **continuos** y **dependientes de la vista**.

La aproximación tradicional, descrita por Clark en 1976, son los denominados LOD discretos. Esto consiste, básicamente, en la simplificación del

modelo para generar varias versiones con diferentes niveles de detalle. Esto se realiza como un preproceso *off-line*, por lo que no afecta al rendimiento de la visualización. Posteriormente, en tiempo de ejecución, se realizará la elección de la versión del modelo a utilizar en cada situación. Esta elección se hará en función de determinados parámetros, generalmente la distancia entre la cámara y el objeto o la carga de *render* del sistema.

En las técnicas de LOD continuos (o progresivos), en lugar de un número finito de versiones del modelo con diferentes detalles, el preproceso genera unas estructuras de datos que codifican un espectro continuo de detalle. A partir de estas estructuras, en tiempo de ejecución se generará el detalle adecuado a cada ocasión, por lo que las transiciones serán más suaves que en el caso de los LOD discretos y, además, la cantidad de detalle utilizado en cada momento será más adecuada, aprovechando mejor los recursos disponibles.

Una mejora sobre los LOD continuos son los denominados LOD dependientes de la vista o de la cámara. En este caso se tiene en cuenta la posición del espectador a la hora de realizar simplificaciones y generar el nivel de detalle en tiempo de ejecución, de forma que un mismo elemento tendrá diferente nivel de detalle en diferentes zonas. Es habitual, por ejemplo, que las zonas cercanas a la cámara o las que conforman la silueta del objeto tengan un mayor detalle. Por este motivo se dice que los LOD dependientes de la vista son **anisotrópicos**.

En el caso de la visualización de terreno, debido a que se trata de modelos de gran tamaño y complejidad, se hace imprescindible el uso de técnicas de LOD dependientes de la vista. El detalle necesario en ciertas zonas cercanas a la cámara debe ser muy superior al de aquellas zonas más lejanas.

La generación y la selección de los LOD se realiza a partir de unos determinados criterios. Estos criterios pueden estar basados en la **fidelidad** de la visualización (por ejemplo, determinando una cota de error máximo) o bien basados en el **coste** o carga del sistema (por ejemplo, determinando un número máximo de polígonos), o una combinación de ambos dando prioridad al que corresponda.

En la selección de niveles de detalle, suele ser habitual utilizar una curva de histéresis para solucionar los problemas derivados de aquellas situaciones en la frontera entre dos niveles de detalle que producen cambios continuos entre uno y otro.

En el campo de la cartografía existe el concepto de **generalización**[111, 106], heredado por los SIG, cuya filosofía es similar a la de los LOD. A grandes rasgos, la generalización en cartografía consiste en el proceso de representar la información en un mapa adaptada a la escala de dicho mapa. El concepto de generalización se retomará más adelante en este trabajo para la representación adecuada de la información SIG.

2.5.3. Técnicas existentes para la visualización de terreno

Nivel de detalle continuo para terreno (Lindstrom)

Lindstrom et al. [103] publicaron en 1996 una de las primeras técnicas de visualización de terreno utilizando LOD continuos para la gestión de una geometría poligonal basada en mallas regulares. Esta técnica permite acotar el error cometido en la imagen final proyectada utilizando un umbral en espacio pantalla. La simplificación de la geometría se realiza en dos pasos. En una primera simplificación por bloques se selecciona el nivel adecuado dentro de un conjunto de LOD discretos. Los LOD elegidos en el paso anterior se retriangulan para refinar el detalle utilizando una técnica de LOD continuo donde se realiza la simplificación por vértice.

Esta técnica ha sido utilizada en numerosos sistemas de visualización de terreno (por ejemplo VGIS[99, 102]) y ha sido adaptada en muchos otros. Sin embargo, en ella no se afronta el problema de la gestión de texturas de gran tamaño para mapear el terreno.

ROAMing (Duchaineau)

En 1997, Duchaineau et al. [66] presentaron la técnica denominada ROAM (Real-time Optimally Adapting Meshes). Esta técnica trata el problema de la gestión de LOD para mallas regulares de geometría. Se basa en una estructura *bin-tree* y en una métrica de error que determina la prioridad de operaciones para unir o separar triángulos de forma que se adapta la malla poligonal en función del punto de vista.

Aunque se plantea la necesidad de un sistema de paginado y selección de LOD para la textura (al igual que la geometría), tampoco se trata este problema. La técnica ROAM ha sido bastante utilizada, especialmente en el campo de los videojuegos. Algunos motores conocidos como Crystal Space, Tread Marks y Genesis3D la implementan.

Rabinovich

En 1997, Rabinovich y Gotsman publicaron una técnica para la visualización interactiva de terreno en entornos computacionales con recursos limitados[123]. El terreno consiste en datos de elevación a partir de una malla regular y textura procedente de imagen aérea o de satélite.

Este trabajo hace especial hincapié en su funcionamiento a través de una red de baja velocidad (3 KBytes/s) y en sistemas de gama baja, tomando una base de datos remota de gran tamaño. Asocian prioridades a los puntos de la malla del terreno para transmitirlos progresivamente según las necesidades de la vista concreta. El cliente mantiene una caché de estos vértices y realiza una triangulación de Delaunay dinámica para *renderizar* la superficie del terreno con los niveles adecuados.

Röttger

Röttger et al. [129] desarrollaron una técnica basada en la de Lindstrom, pero utilizando estructuras *quadtree* en lugar de *bintree* y una aproximación descendente en lugar de la ascendente de Lindstrom. En las pruebas del artículo utilizan una textura única para mapear toda la geometría, por lo que están limitados por el tamaño máximo de textura soportado por el *hardware* gráfico.

Mallas progresivas (Hoppe)

Hoppe propone en 1998 una técnica, denominada "mallas progresivas" o PM (*progressive meshes*)[91] que utiliza mallas irregulares (TIN). Esta técnica se basa en otro trabajo suyo, denominado "malla progresiva dependiente de la vista" o VDPM [90] (*view dependent progressive mesh*), especializándolo para el caso concreto de la visualización de terreno.

Para grandes extensiones de terreno, se realiza un preproceso que particiona la geometría en bloques organizados en una estructura jerárquica. Cada uno de estos bloques será una malla progresiva (PM) basada en TIN.

Para la gestión de la textura plantea dos alternativas: el uso de *clip-mapping*[144] (cuando se disponga de *hardware* que lo soporte) o la división de la textura del terreno en fragmentos (*tiles*) asociados uno a uno con los bloques de geometría (PM). Esta segunda opción es uno de los motivos por los que es necesaria la partición del terreno en bloques.

Multitriangulación (De Floriani y Puppo)

Otra técnica de manejo de LOD geometría basada en TIN, propuesta en 1998, es la multitriangulación (MT) [73] desarrollada por Leila De Floriani y Enrico Puppo. Esta técnica es genérica y se puede aplicar a terreno así como a otros tipos de mallas poligonales. Los autores se limitan al modelo geométrico y no consideran el tema del mapeado de texturas en absoluto.

GeoMipmaps (De Boer)

En el año 2000, Willem H. De Boer publicó una técnica denominada GeoMipMapping[61]. De Boer plantea que las técnicas existentes no tienen en cuenta las capacidades del *hardware* gráfico actual y que buscan la "geometría perfecta" midiéndola en factores como el número de triángulos, cuando lo importante no es la cantidad de polígonos que se pasen al *pipeline* gráfico sino que se pasen en una forma adecuada para optimizar el rendimiento y liberar la carga de la CPU. Es decir, es preferible dibujar más triángulos si esto implica conseguir un tiempo de *render* inferior.

La técnica planteada por De Boer aplica el concepto de *mipmap*[148] al campo de la geometría. Siguiendo esta analogía con el filtrado trilineal de

las texturas, gestiona los LOD de una geometría basada en mallas regulares. Sin embargo, De Boer no menciona ninguna técnica para aplicar textura a la geometría manejada mediante GeoMipMaps.

Lindstrom y Pascucci

Lindstrom y Pascucci publicaron en el año 2001 una técnica [104] basada en mallas regulares y *bintrees* cuyo objetivo principal era la sencillez en sus algoritmos y estructuras de datos para conseguir un buen rendimiento, separándose de la tendencia existente hacia los algoritmos cada vez más complicados. Proporciona un refinamiento de grandes superficies de terreno en función de la vista y gestiona la paginación para manejar volúmenes de datos que superen la memoria disponible (*out-of-core visualization*). De nuevo, en esta técnica los autores se limitan a la gestión del modelo geométrico del terreno, sin tocar para nada el texturizado.

Klein y Schilling

Reinhard Klein y Andreas G. Schilling [98] publicaron en 2002 una técnica de visualización en tiempo real de terreno multirresolución basada en una partición y organización en una estructura jerárquica de tipo *quadtree* adecuada para su transmisión progresiva.

Esta técnica incluye la gestión de la textura del terreno, tal y como se describe más adelante.

BDAM/P-BDAM (Cignoni et al.)

En el año 2003, Cignoni et al. publicaron el algoritmo BDAM (*Batched Dynamic Adaptive Meshes*) [49] extendido posteriormente a escala planetaria en el P-BDAM (*Planet-Sized BDAM*) [50].

Combina las ventajas de TIN y HRT (*Hierarchical Right Triangles*), un tipo de estructura regular jerárquica. Evitan el cuello de botella entre la CPU y la GPU debido al elevado número de polígonos mediante la gestión de la geometría en paquetes de triángulos preprocesados off-line y organizados en estructuras eficientes como tiras de triángulos (*tristrips*). De esta forma varían el grano de los modelos multirresolución de triángulos a pequeñas mallas. Los HRT son organizados jerárquicamente en una estructura *bintree*.

La técnica de BDAM gestiona también la textura del terreno, tal y como se describe más adelante.

Seoane

Otra aproximación a la gestión de terreno fue desarrollada el año 2004 en la Universidade da Coruña por Antonio Seoane [136]. Esta técnica gestiona los LODs del terreno y la paginación de bases de datos cuyo tamaño desborda

las capacidades del sistema. Se basa en una estructura de caché a dos niveles (memoria principal y disco).

La gestión de la geometría está basada en la técnica propuesta por De Boer [61], con algunas mejoras como la posibilidad de combinar LODs con un salto de varios niveles en parcelas de terreno contiguas manteniendo las uniones correctamente. Este trabajo también describe una técnica de gestión de texturas de terreno, basada en una implementación de *clipmapping*[144] sin requisitos particulares de *hardware*[137], tal y como se describe más adelante.

Geometry clipmaps (Losasso y Hoppe)

En el SIGGRAPH de 2004, Frank Losasso y Hugues Hoppe presentaron una técnica de gestión de geometría denominada *Geometry Clipmaps* [107], que como su nombre indica, se basa en la técnica de Tanner et al. [144] para la gestión de texturas de tamaño virtualmente ilimitado, descrita en la sección 2.6.3.

Mantienen una caché de LODs de geometría anidados en torno a un centro de detalle. Cada LOD corresponde a una matriz cuadrada de igual número de vértices (y mayor extensión geográfica según se reduce la resolución geométrica). Esta caché se va actualizando según se desplaza la cámara y el sistema aplica la geometría disponible en la memoria gráfica, que debería ser un subconjunto de la geometría total adecuada para *renderizar* la vista actual sin un error geométrico importante.

Para evitar huecos en las uniones entre LODs, realizan una deformación progresiva (*morph*) a lo largo de una zona de transición.

Utilizan VBOs (*vertex buffer objects*) para almacenar la caché de geometría en la memoria del sistema gráfico. Y sobre el detalle geométrico disponible, procedente de la base de datos, generan detalle adicional mediante técnicas de ruido fractal para dar un mayor realismo visual a la escena.

Junto con cada nivel del *clipmap* de geometría, se tiene una textura asociada. En la implementación de los autores, esta textura se utiliza como mapa de normales para el terreno.

En el año 2005, Arul Asirvatham y Hugues Hoppe publicaron una implementación de esta técnica basada en GPU utilizando la reciente capacidad de acceder a texturas desde los *vertex shaders* [37].

Holkner

Alex Holkner publicó, también en 2004, otra técnica similar a la de Losasso y Hoppe, trasladando el concepto de *clipmap* a la gestión de geometría de terreno [89]. La implementación que propone está basada en GPU, siguiendo en esencia la misma idea que publicarían posteriormente Asirvatham

y Hoppe, de utilizar texturas para almacenar los vértices de los niveles del *clipmap* y utilizar estas texturas desde un *vertex shader*.

Un detalle interesante que indica Holkner, diferenciando el uso de *clipmaps* para geometría de su uso original para texturas, hace referencia al filtrado de los niveles de detalle. En las texturas es habitual filtrar utilizando un núcleo de convolución uniforme (*box filter*) con resultados aceptables, sin embargo en el caso particular de la geometría, los resultados no son adecuados, puesto que produce efectos como aplanar grandes montañas y suavizar detalles. Por este motivo utilizan la función seno cardinal (o sinc) como núcleo de convolución con una ventana de Blackman, lo cual conserva mucho mejor los picos y valles del terreno.

2.6. Sistemas de texturizado de terreno

2.6.1. Mapeado de texturas

El mapeado de texturas [48] o texturizado es una técnica utilizada desde hace décadas para añadir detalle o mejorar el aspecto de los modelos geométricos sin incrementar su complejidad.

Esta técnica consiste en aplicar una imagen sobre los modelos geométricos de la escena visualizada. Esta imagen modifica el aspecto de la superficie de esos modelos de diversas maneras. Una forma intuitiva de describir el uso habitual del texturizado es como un "papel de pared" digital que se pega sobre la superficie de los objetos.

La imagen está formada por una matriz bidimensional[12] de celdas denominadas **texels** (elementos de textura o celdas de textura). Los valores de los texels representan parámetros que varían a lo largo de la superficie, típicamente intensidad, color, opacidad o cualquier otro parámetro (en este caso se suelen denominar mapas temáticos) como tipo de suelo, temperatura, etc. El contenido de la imagen se puede aplicar directamente a la superficie como color y opacidad o combinar con los cálculos de iluminación en esa superficie, aunque hay muchas y muy variadas aplicaciones[86, 85].

La imagen aplicada sobre la geometría se denomina habitualmente "imagen de textura", "mapa de textura" o simplemente "textura", aunque en gran parte de los casos no se trata de una textura propiamente dicha. El nombre surge originalmente de las imágenes que se aplicaban repetidas (formando un mosaico) para simular el aspecto de un material como madera, marmol, tela, etc. En la actualidad se utiliza este nombre igualmente, aunque se trate, por ejemplo, de la fotografía de una fachada aplicada sobre un modelo 3D del edificio o una imagen plana aplicada sobre un cartel (*billboard*).

[12] Aunque es el caso más habitual, las texturas no necesariamente son bidimensionales. De hecho, el *hardware* gráfico actual soporta texturas 1D, 2D o 3D. Sin embargo, en este trabajo nos centraremos en la aplicación de texturas formadas por imágenes bidimensionales.

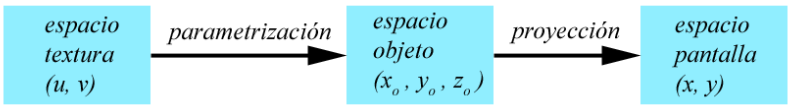


Figura 2.19: Conversión de espacios de coordenadas en el mapeado de texturas.

El mapeado de texturas se puede dividir en dos fases[86, 87]: El **mapeado** geométrico que adapta la imagen bidimensional a la superficie tridimensional y el **filtrado** necesario para evitar artefactos en la visualización.

El proceso de mapeado propiamente dicho consiste en la transformación entre espacios de coordenadas. La imagen original (espacio textura) se mapea sobre un objeto tridimensional (espacio objeto) que a su vez es mapeado en la pantalla (espacio pantalla) como resultado de una proyección, que podrá ser ortográfica o, más habitualmente, perspectiva[74]. Utilizaremos (u, v) para representar las coordenadas 2D en espacio textura, (x_o, y_o, z_o) para representar las coordenadas 3D en espacio objeto y (x, y) para representar las coordenadas 2D en espacio pantalla. La conversión de espacio textura a espacio objeto se denomina **parametrización** y de espacio objeto a espacio pantalla **proyección** (figura 2.19). El proceso de proyección de espacio objeto a espacio pantalla consiste realmente en las transformaciones de modelo y vista seguidas de la proyección. Sin embargo, todas estas transformaciones, incluyendo la parametrización, se suelen combinar en una sola y desde el punto de vista del mapeado de texturas se considera directamente la transformación de espacio textura a espacio pantalla, ignorando todos los pasos intermedios.

La relación del mapa de textura y la superficie sobre la que se aplica se puede realizar a través de funciones de proyección o definiendo explícitamente las coordenadas de textura.

En los modelos consistentes en mallas poligonales, que son los que se utilizan en la visualización en tiempo real, cada vértice puede tener asociado un vector de coordenadas de textura, que asigna a la posición de ese vértice en espacio objeto (x_o, y_o, z_o) un punto en el espacio de coordenadas de textura (u, v). En el interior de los polígonos, las coordenadas de textura se calculan mediante la interpolación de las coordenadas de textura de los vértices de dichos polígonos. En ocasiones, las coordenadas de textura de los vértices no se definen explícitamente sino que se calculan automáticamente a partir de ciertas proyecciones (plana, cilíndrica, esférica, cúbica).

En el *render* en tiempo real, las coordenadas de textura de cada *fragment* se obtienen mediante la interpolación de las coordenadas de textura de los vértices del triángulo que contiene el *fragment*. Estas coordenadas de textura sirven para direccionar una posición dentro de la imagen de la textura, realizando las operaciones descritas en la figura 2.20 y aplicando el resultado

al *fragment* que se está procesando.

Es importante indicar que en OpenGL[135], las coordenadas de textura asociadas a los vértices de un modelo geométrico, se definen en un espacio de coordenadas normalizado y se suelen representar como (s,t). Para convertir a coordenadas (u,v) en el espacio textura, simplemente se multiplican por el tamaño (en texels) de la imagen de la textura (ecuación 2.6). En muchas ocasiones, en la literatura se referencian las coordenadas de textura como (u,v) o (s,t) indistintamente. En esta memoria se seguirá el criterio indicado, utilizando (u,v) para el espacio textura en texels y (s,t) para el espacio textura normalizado.

Las coordenadas de textura (s,t) en OpenGL pueden contener valores fuera del intervalo $[0,1]$. En este caso, la correspondencia en espacio textura (u,v) estará determinada por la configuración de la máquina de estados de OpenGL, pudiendo repetirse cíclicamente la textura, repetirse invertida (en espejo), tomarse un valor fijo, el valor del texel más cercano, etc. Además, las coordenadas de textura están afectadas por una matriz de transformación, que puede aplicar traslaciones, rotaciones y escalados a la posición de la imagen sobre la superficie mapeada.

Tras el proceso de mapeado, descrito en la figura 2.19, y con la imagen ya deformada para adaptarla a su posición en espacio pantalla, se realizan los procesos de filtrado con objeto de eliminar efectos indeseables denominados *aliasing*, que se describen más adelante.

Desde el punto de vista del tratamiento de imágenes, el proceso de mapeado de una textura (figura 2.20) consiste en una reconstrucción de la imagen muestreada en la textura[13], una deformación de dicha imagen reconstruida para adaptarla a la forma de la geometría en espacio pantalla (el proceso de mapeado propiamente dicho), un filtrado (paso bajo) de la imagen deformada para eliminar componentes de alta frecuencia que puedan causar *aliasing* (como se describe más adelante) y finalmente una remuestreo de la imagen reconstruida, deformada y filtrada[86, 135]. Tras estos procesos se obtiene el valor final de la textura para aplicar a cada *fragment*.

Aliasing y filtrado de las texturas

El fenómeno de *aliasing* se produce en muchos campos del procesado digital de señales (del cual las imágenes son un caso particular: señales en 2D), afectando negativamente a la calidad de las señales reconstruidas. Su causa es el muestreo de una señal a una frecuencia insuficiente. Según el teorema de muestreo de Nyquist-Shannon[119, 138], para reconstruir correctamente una señal, ésta debe ser muestreada a una frecuencia igual o superior al doble de la frecuencia máxima de dicha señal. Puesto que una imagen habitualmente no tiene un límite máximo a las frecuencias (el borde de un objeto donde hay

[13] La imagen de la textura no es más que el resultado de un muestreo de una información continua a una frecuencia (i.e. una resolución) determinada.

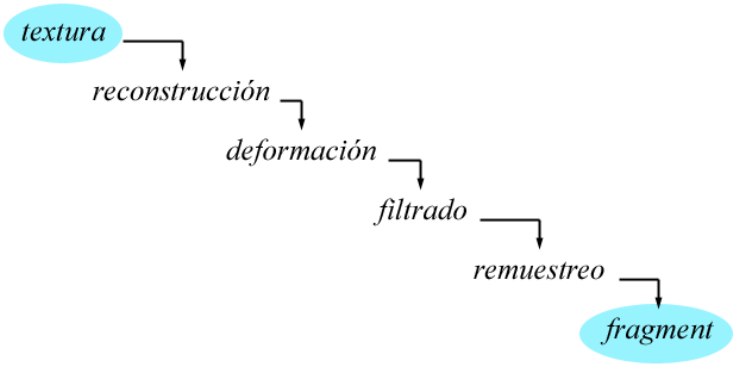


Figura 2.20: Proceso de mapeado de texturas.

un cambio instantáneo de color supone una frecuencia infinita), el fenómeno de *aliasing* casi siempre va a estar presente en las imágenes digitales. Sin embargo, se pueden aplicar técnicas que reduzcan la pérdida de calidad en la imagen o la animación final.

Sin entrar a profundizar en la teoría de señales, podemos mencionar que las consecuencias del *aliasing* con respecto al mapeado de texturas se materializan en parpadeos, escaleras o dientes de sierra, patrones *moiré* o ruido en general (figuras 2.23 y 2.25). Estos efectos, visibles claramente en una imagen estática, se hacen mucho más notables cuando se muestra una animación, especialmente si los objetos se mueven lentamente en pantalla.

En el caso de las texturas, el problema surge cuando la correspondencia entre los texels (en el espacio de imagen de la textura) y los pixels (en el espacio de la pantalla) no es 1:1. Cuanto más se aleje de esta proporción, más grave será el efecto. En este caso se pueden dar dos situaciones: que los texels sean más grandes que los pixels o que sean más pequeños. En el primer caso se produce un aumento de la textura y en el segundo una reducción respecto a su resolución original.

Para evitar el problema de *aliasing*, se utilizan las técnicas denominadas *anti-aliasing*. En el caso del mapeado de texturas, se realiza un filtrado de la textura (ver figura 2.20). El tipo de filtro a utilizar dependerá de si se realiza un aumento o una reducción de la textura en espacio pantalla.

En OpenGL[135], el tipo de filtrado a utilizar se decide a partir de un parámetro denominado **nivel de detalle** ($\lambda(x, y)$), que se define, a grandes rasgos[14], en las ecuaciones 2.3 y 2.4.

[14]En realidad, los cálculos son un poco más complejos. El valor λ puede tener aplicados varios valores de sesgo asociados a la textura, a la unidad de texturizado o al *shader* y ajustados a un máximo en el valor absoluto del sesgo. Para más información sobre las especificaciones exactas de OpenGL, consultar [135].

$$\lambda'(x,y) = \log_2[\rho(x,y)] \tag{2.3}$$

El valor de nivel de detalle λ' calculado en la ecuación 2.3 se ajusta a unos límites máximo y mínimo establecidos en la máquina de estados de OpenGL.

$$\lambda = \begin{cases} lod_{max} & \lambda' > lod_{max} \\ \lambda' & lod_{min} \leq \lambda' \leq lod_{max} \\ lod_{min} & \lambda' < lod_{min} \end{cases} \tag{2.4}$$

El valor de λ determina si se utilizará un filtro de aumento (*magnification*) o reducción (*minification*), según sea menor o igual o mayor que la constante c[15] respectivamente.

El valor ρ a partir del cual se calcula λ se denomina **factor de escala**, y se calcula a partir de las derivadas parciales de los parámetros (u,v) (las coordenadas en el espacio de imagen de textura) respecto a (x,y) (las coordenadas en espacio pantalla), como se describe en la ecuación 2.5.

$$\rho = \text{máx}\left(\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2}\right) \tag{2.5}$$

donde

$$\begin{aligned} u(x,y) &= w_t \times s(x,y) \\ v(x,y) &= h_t \times t(x,y) \end{aligned} \tag{2.6}$$

siendo s y t las funciones que asocian la coordenada de textura (en el espacio de textura normalizado) correspondiente con las coordenadas de pantalla (x,y) dentro de una primitiva y w_t y h_t los tamaños (en texels) en cada dimensión (ancho, alto) de la textura (incluyendo los pixels del borde en caso de usarlo).

Aumento. Filtrado por proximidad y bilineal

En caso de aumento de la textura, en OpenGL se pueden utilizar dos tipos de filtro: proximidad y bilineal.

El filtro de **proximidad** consiste en tomar una única muestra en el texel de la textura dentro del que caen las coordenadas de textura. En el ejemplo de la figura 2.21, la muestra que se tomaría para las coorenadas de textura (u,v) sería (u_1, v_2).

La segunda opción es el filtrado **bilineal**, que consiste en tomar cuatro muestras en los cuatro texels que rodean al punto de la textura correspondiente a las coordenadas de textura y realizar tres interpolaciones lineales

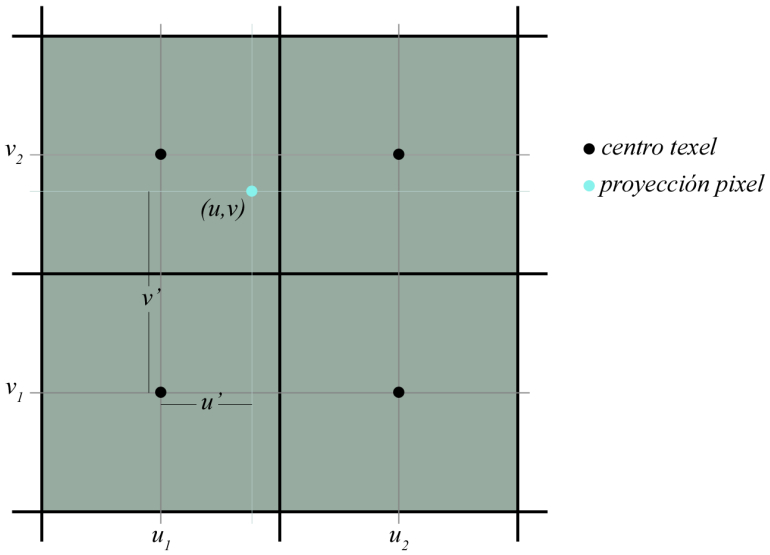


Figura 2.21: Filtrado bilineal.

tal y como se describe en la ecuación 2.7 y la figura 2.21.

$$\begin{array}{lcl} u' & = & u - \lfloor u \rfloor \\ v' & = & v - \lfloor v \rfloor \\ u_1 = \lfloor u \rfloor & & \\ u_2 = u_1 + 1 & & \\ v_1 = \lfloor v \rfloor & & \\ v_2 = v_1 + 1 & & \\ b(u,v) & = & (1-u')(1-v')t(u_1,v_1) + u'(1-v')t(u_2,v_1) \\ & & +(1-u')v't(u_1,v_2) + u'v't(u_2,v_2) \end{array} \tag{2.7}$$

El primer tipo de filtrado es el menos costoso en tiempo de cálculo, pues simplemente se toma una única muestra y se utiliza directamente como el valor para ese *fragment*. Sin embargo, también es el que produce un mayor *aliasing*, de forma que al aumentar la textura la imagen aparece pixelada (ver figura 2.22a). El filtrado bilineal mejora estos resultados, suavizando las transiciones entre texels, como se aprecia en la figura 2.22b. El *hardware* gráfico actual implementa este filtrado de forma muy eficiente, por lo que en la mayor parte de las ocasiones no existe motivo para no utilizarlo.

El filtrado bilineal no es la mejor opción para aumentar la textura, pero se trata de una aproximación muy eficiente para utilizar en tiempo real. En los programas de procesamiento de imágenes suele haber disponibles otros filtros

[15] Esta constante tomará valor 0 o 0,5 según los filtros de aumento y reducción que se hayan configurado (consultar [135]).

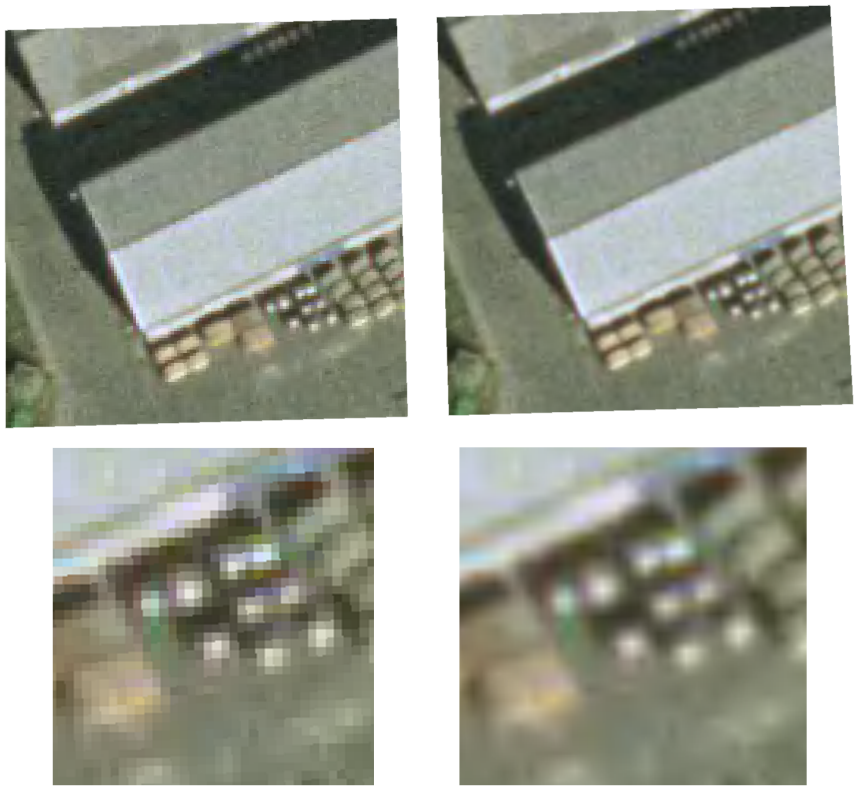


Figura 2.22: Comparativa de técnicas de filtrado para el aumento de la textura. a) Textura aumentada con filtrado por proximidad (*nearest*) b) Textura aumentada con filtrado bilineal.

que ofrecen mayor calidad (bicúbico, gaussiano, lanczos, etc.), sin embargo, el coste en relación al aumento de calidad que suponen estos filtrados hace que no se utilicen para las aplicaciones en tiempo real.

Reducción. *Mipmapping*

En el caso de reducción de la textura el efecto producido cuando se toma una única muestra es incluso peor, sobre todo en una animación donde la geometría texturizada se desplaza lentamente por la pantalla. El problema es debido a que la zona correspondiente a un pixel en pantalla cubre muchos texels en la textura. Si se toma una única muestra por proximidad, se saltan muchos texels intermedios que deberían contribuir al valor final del *fragment*. Además, al desplazarse levemente el objeto mapeado o la cámara, se saltará de un texel a otro produciendo parpadeos, vibraciones, escaleras, efectos *moiré* (figura 2.23) y en el mejor de los casos un molesto ruido e inestabilidad en la imagen.

Para reducir el problema de *aliasing* en la reducción de texturas, se

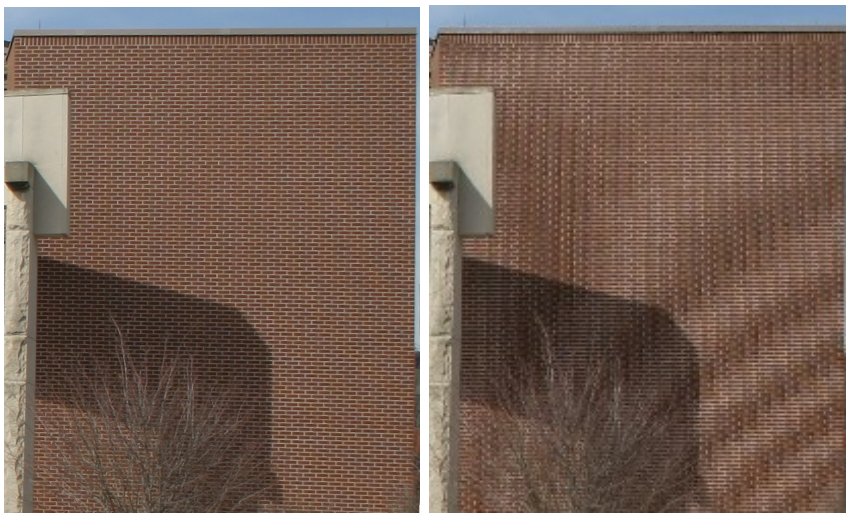


Figura 2.23: Efecto *moiré* producido por un remuestreo reduciendo la imagen original tomando muestras por el método de proximidad. A la izquierda la imagen original y a la derecha la versión reducida.

pueden seguir dos estrategias diferentes[86]: muestrear a una resolución mayor o aplicar un filtro paso bajo antes de muestrear.

La primera de las soluciones implica tener en cuenta muchos texels para calcular el valor de un pixel. Tomar muchas muestras en la textura implica realizar gran cantidad de costosas lecturas de memoria, además de los cálculos aritméticos necesarios para realizar la convolución, lo cual provoca un incremento en el coste de cálculo en los filtros de reducción, que en situaciones donde la cámara se situa de forma oblícua a la superficie texturizada llega a ser absolutamente inviable para aplicaciones interactivas.

Por este motivo, la segunda aproximación resulta más adecuada, especialmente para aplicaciones en tiempo real. Según Heckbert, esta aproximación se enfrenta a la causa del *aliasing* en lugar de a los síntomas[86]. La imagen se limita en frecuencia, de forma que antes de muestrearla nos aseguramos de que no existan frecuencias por encima del límite impuesto por el teorema de Nyquist.

La solución más extendida para reducir el *aliasing* de las texturas en el *render* en tiempo real es el uso de una estructura piramidal denominada ***mipmap*** o ***MIP map***[16], sugerida por William Dungan en 1978 [147] y desarrollada y popularizada posteriormente por Lance Williams en 1983 [148], que fue realmente quien acuñó el término. El *mipmap* se compone de la imagen original correspondiente a la textura junto con versiones reducidas (prefiltradas) de dicha imagen (ver figura 2.24). Cada nueva imagen de la

[16]MIP viene del latín *multum in parvo* (muchas cosas en un lugar pequeño), haciendo referencia a que en un pixel se almacena la información correspondiente a un conjunto potencialmente numeroso de pixels en la resolución original de la imagen.

Figura 2.24: Estructura piramidal con los diferentes niveles de detalle prefiltrados de una textura (*mipmap*).

textura está reducida a la mitad de resolución en cada eje (es decir, a la cuarta parte). De esta forma, según el factor de reducción (λ) necesario en el momento de mapear la textura en un *fragment*, se tomarán los niveles de detalle correspondientes que ya tendrán precalculada la convolución de todos los pixels de la zona correspondiente. Esto facilita el trabajo de filtrado de forma que se puede solucionar de modo eficiente con interpolaciones lineales inter- e intra-nivel.

Todo el *hardware* gráfico disponible en la actualidad soporta el uso de *mipmaps* para el filtrado en la reducción de texturas. Por motivos evidentes, el uso de texturas con tamaños potencia de dos facilita tanto la generación de los niveles del *mipmap* como su posterior mapeado. No obstante, las últimas generaciones de *hardware* gráfico soportan el uso de texturas de tamaño no potencia de dos.

En OpenGL, el filtrado para la reducción de la textura utilizando *mipmap*, se realiza calculando el nivel de detalle (λ), que será un número real comprendido entre dos niveles del *mipmap*. En cada uno de estos niveles, que deberían contener una versión prefiltrada a la escala correcta, se toman las cuatro muestras que rodean la posición determinada por las coordenadas de textura, y se realiza un filtrado bilineal, tal y como se describió en la ecuación 2.7 y la figura 2.21. Los resultados obtenidos en cada uno de los niveles, se interpolan de forma lineal para obtener el resultado final. Por este motivo, este filtrado se denomina **trilineal**, y en general proporciona un resultado bastante aceptable y eficiente para su uso en tiempo real.

Es posible también utilizar versiones más sencillas, mediante el muestreo por proximidad en lugar de la interpolación lineal, tanto para elegir el nivel del *mipmap* como para calcular el valor dentro de el o los niveles escogidos. No utilizando la estructura de *mipmap*, las opciones se reducen a las mismas que en el filtrado de aumento: bilineal o proximidad.

Un detalle importante del uso de *mipmap* es que el filtrado se realiza en un preproceso y el resultado se carga en memoria de texturas. Por este motivo se puede utilizar cualquier técnica para el filtrado, no estando limitados a unos filtros concretos. El tiempo de cálculo no es crítico puesto que sólo se realizará una vez en el momento de generación de las texturas y no afectará

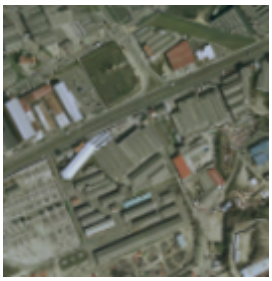
a)  b)  c) 

Figura 2.25: Comparativa de técnicas de filtrado para la reducción de la textura. a) Filtrado por proximidad b) Fitrado bilineal c) Filtrado trilineal (*mipmap*).

a su uso en tiempo de ejecución.

Además del *mipmap*, existen otras técnicas para filtrar las texturas y reducir el *aliasing* de forma eficiente, como las propuestas por Norton[117] y Crow[57]. Sin embargo, el *mipmap* es la técnica que ha tenido más aceptación y que se encuentra implementada en la práctica totalidad del *hardware* gráfico disponible en la actualidad, así como los APIs gráficos más extendidos (OpenGL y DirectX).

Filtrado anisotrópico

El problema de los *mipmap* es que se asume que cada pixel corresponde a una zona cuadrada de la textura, y por tanto que la compresión de la textura es simétrica[57]. Dicho en otras palabras, se presupone que la textura se escala de igual manera en ambos ejes.

Sin embargo, esto sólo ocurre si observamos la superficie mapeada en la dirección de su vector normal. En cuanto se varía el ángulo de la cámara respecto a esa superficie, se comprime más la textura en un sentido que en otro. Se produce por tanto una situación de **anisotropía**.

La técnica de *mipmap*, adecuada únicamente para situaciones isotrópicas, considera la mayor de las derivadas de los dos ejes de la textura respecto a los dos ejes de la pantalla para calcular el factor de escala, tal y como se describe en la ecuación 2.5. Es decir, se necesita un área rectangular de la textura, pero el *mipmap* sólo proporciona regiones cuadradas en diferentes niveles de detalle. Por lo tanto, cuando hay diferencia en ambos ejes, uno de ellos aparecerá excesivamente borroso para evitar el *aliasing* en el otro.

Esta situación se produce con mucha frecuencia en el caso de la visualización de terreno, puesto que es muy habitual que la vista se dirija hacia el horizonte. En la figura 2.26a se ilustra este efecto sobre la pista de un aeropuerto.

Existen diferentes soluciones a este problema, denominadas algoritmos de filtrado anisotrópico. A continuación se describen algunas de ellas.

Los ***ripmaps*** son una extensión de los *mipmaps* que añaden areas rec-

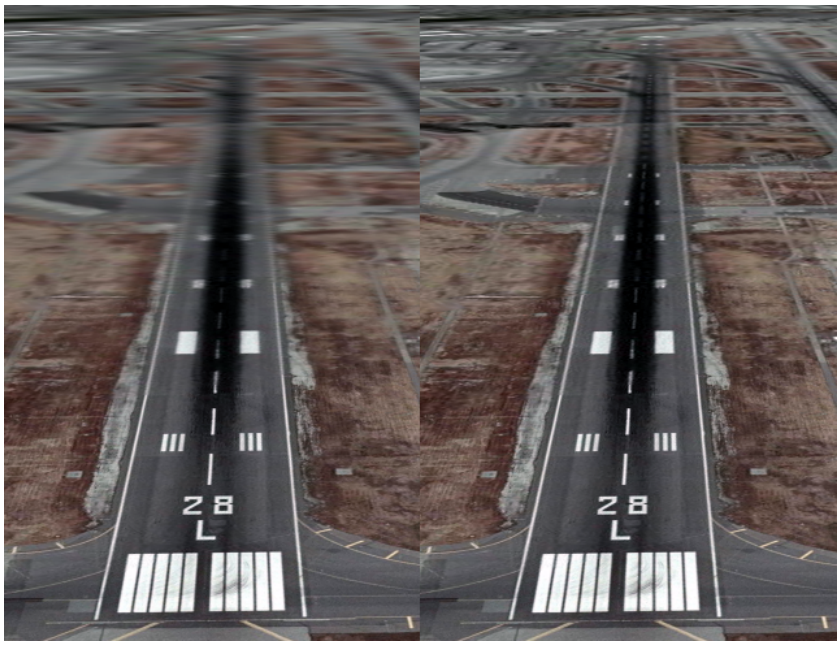


Figura 2.26: Efecto borroso al aplicar la técnica de *mipmap* en una situación de anisotropía (izquierda) y versión corregida mediante filtrado anisotrópico (derecha).

tangulares submuestreadas además de las cuadradas (figura 2.27). De esta forma, al elegir las muestras para interpolar, se determinan los mapas adecuados según el área de la textura ocupada por el pixel a *renderizar*. En lugar de tres coordenadas (u, v, λ), se establecen cuatro, las dos últimas que determinan el o los niveles a elegir, estarán calculadas en función de cuánto se extiende el pixel sobre la textura en u y v. La principal desventaja de los *ripmaps* es que utilizan mucha memoria, recurso crítico en el *hardware* gráfico. Esta técnica se implementó en algunas aceleradoras gráficas de gama alta de Hewlett-Packard a principios de los 90.

Otra técnica de filtrado anisotrópico son las **tablas de suma de áreas**[57], desarrollada por Franklin C. Crow en 1984. En lugar de la pirámide de *mipmap*, se utiliza una única tabla del tamaño total de la textura donde en cada celda, en lugar del valor del texel, se almacena la suma de todos los texels desde el $(0, 0)$, situado en la esquina inferior izquierda, hasta la posición de dicho texel. De esta forma, para obtener el valor filtrado correspondiente a cualquier área rectangular de la textura, limitada por u_l a la izquierda, u_r a la derecha, v_b abajo y v_t arriba, se toma el área hasta la esquina superior derecha, y se le restan las áreas de las esquinas superior izquierda e inferior derecha. Como el área correspondiente a la esquina inferior izquierda se ha restado dos veces, se vuelve a sumar. Finalmente, se divide el resultado por el área calculada para obtener el promedio de los valores de la textura en esa zona. Estas operaciones se resumen en la ecuación 2.8 y se ilustran en la

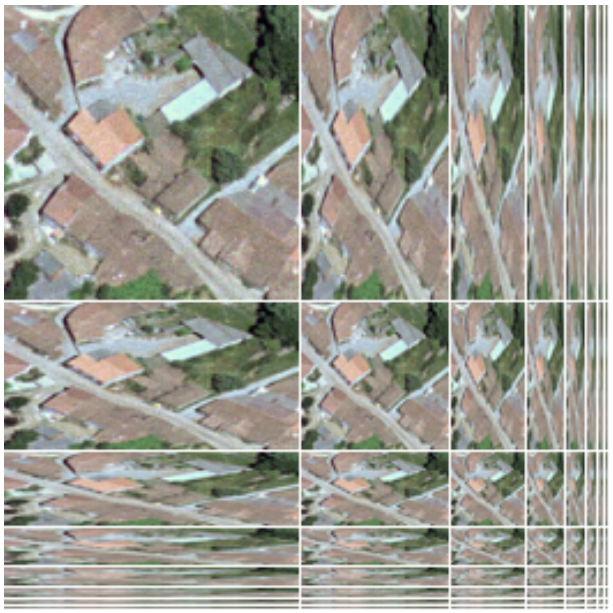


Figura 2.27: *Ripmap*.

Técnica de filtrado	Requerimiento de memoria total
Mipmap	133 %
Ripmap	400 %
Tabla de suma de área	200 %+ en función del tamaño de la textura[17]

Cuadro 2.5: Comparativa de uso de memoria de las diferentes técnicas de filtrado de reducción de texturas.

figura 2.28.

$$\frac{T(u_r, v_t) - T(u_r, v_b) - T(u_l, v_t) + T(u_l, v_b)}{(u_r - v_l)(u_t - v_b)} \tag{2.8}$$

El problema de las tablas de suma de áreas es que aunque el número de celdas corresponde con el número de texels, los valores almacenados en esas celdas, pueden tener magnitudes enormes, por lo que la resolución en bits de la información almacenada por cada celda es mucho mayor que la de la imagen original. Este problema crece exponencialmente con el tamaño de la textura. Por este motivo, esta técnica, a diferencia de los *ripmaps*, no ha llegado nunca a ser implementada en *hardware*. Puesto que el *hardware* gráfico de consumo desde 2004 incluye la posibilidad de trabajar en aritmética en coma flotante, una posibilidad sería utilizar este tipo de datos para almacenar los valores de la tabla, sin embargo, esto provocaría una pérdida de precisión en los valores de la textura, sobre todo cuanto más detallado sea el nivel que se utiliza.

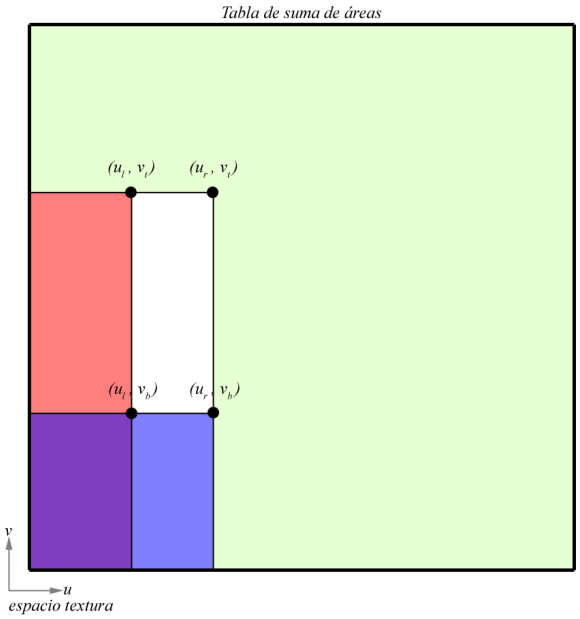


Figura 2.28: Esquema de funcionamiento de la técnica de tabla de suma de área.

Otro problema que sufren las dos técnicas descritas es que están preparadas para calcular con cierta precisión la zona rectangular cubierta por un pixel en el espacio de la textura siempre que esté alineada con los ejes de ésta. Esta zona se denomina habitualmente en la literatura sobre el tema ***footprint* del pixel**[147]. Si el *footprint* del pixel está situado en diagonal (figura 2.29), y especialmente cuando se trata de un rectángulo estrecho, el área de la textura que se convoluciona cubre muchos más texels de los que realmente corresponden a ese pixel, lo que resultará en valores incorrectos en la imagen final, que resultará excesivamente borrosa.

Debido a las limitaciones y requisitos de las otras aproximaciones, las solución más comunmente implementada en el *hardware* gráfico actual, resuelve el problema de la anisotropía basándose en los mecanismos de *mipmapping* ya existentes en dicho *hardware*. La zona correspondiente a un pixel se puede aproximar mediante un rectángulo en espacio textura. Este rectángulo se reconstruye a partir de varias muestras cuadradas del *mipmap*, aproximando el área del rectángulo a partir de cuadrados.

El lado menor de este rectángulo se utiliza para calcular el nivel (o los niveles) de la pirámide que se utilizará(n), y el lado mayor se utiliza para definir una **línea de anisotropía**, paralela a este lado y que pasa por el centro del rectángulo (ver figura 2.29). A lo largo de esta línea se tomarán el número de muestras que resulte adecuado según la **ratio de anisotropía** (el cociente entre el lado mayor y el lado menor).

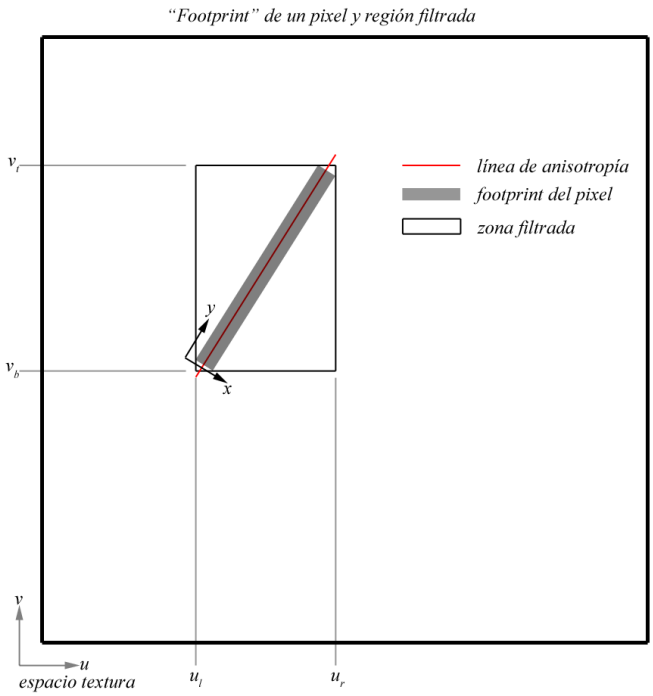


Figura 2.29: Gestión del área de un pixel en espacio textura por parte de las técnicas de filtrado anisotrópico.

De esta forma, se resuelven los problemas de consumo de memoria de técnicas como los *ripmaps* o las tablas de suma de áreas así como la adaptación a cualquier orientación de anisotropía, y no sólo a aquellos casos en que está alineada con los ejes del espacio textura.

Según aumenta el número de muestras de *mipmap* a tomar para resolver la anisotropía, se aumenta proporcionalmente el tiempo de cálculo de los *fragments*. El *hardware* gráfico puede paliar este problema incluyendo varios subsistemas de texturizado que permitan realizar otros tantos accesos simultáneos, de forma que el rendimiento no se vea afectado. En estos casos, la pérdida de rendimiento irá en función del múltiplo del número de subsistemas de texturizado al que corresponda el número de muestras tomadas.

Las técnicas para solucionar el problema del filtrado muestreando a lo largo de cualquier eje ya se plantean en el trabajo de Catmull en 1974[48] y van evolucionando con los trabajos de James F. Blinn y Martin E. Newell (1976) [41], Eliot A. Feibush, Marc Levoy y Robert L. Cook (1980) [71] y Didier Perny, Michel Gangnet y Philippe Coueignoux (1982) [122].

Andrew Glassner propuso en 1986 una técnica adaptativa e iterativa, basada en la tabla de suma de áreas de Crow, para obtener una precisión arbitrariamente alta, a costa de un tiempo de cómputo considerablemente elevado en relación a las otras técnicas.

Greene y Heckbert retoman el tema en 1986 con un buen estudio del estado del arte y plantean el uso de un filtro de Gauss de forma elíptica, denominado EWA (*Eliptical Weighted Average*)[81, 86, 87]. Este algoritmo es uno de los que ofrece una mejor calidad de los disponibles hasta la actualidad. El motivo para utilizar un filtro de Gauss en lugar de la función *sinc* (que teóricamente es el filtro perfecto para reconstruir la señal) es que decae más rápidamente y por lo tanto se pueden obtener buenos resultados con un núcleo de convolución de inferior tamaño.

Trabajos posteriores en la segunda mitad de la década de los 90 resuelven el tema del filtrado anisotrópico mediante diseños de *hardware* específico. Entre ellos se pueden mencionar el sistema Texram de Schilling et al. [131], el Talisman de Barkans et al. [40] y el Feline de McCormack et al. [109]. Es muy interesante y significativo el hecho de que todos estos algoritmos pensados para implementar en *hardware* se basan en calcular la línea y la ratio de anisotropía para posteriormente tomar varias muestras utilizando el *hardware* de filtrado trilineal basado en *mipmaps*, tal y como se ha descrito.

Una variación del Texram fue propuesta por Tobias Hüttner y Wolfgang Strasser en 1999 [93], que también está basada en el uso del *hardware* de filtrado trilineal mediante *mipmaps* con algunas modificaciones para combinar varios texels filtrados trilinealmente sin superar un límite de muestras impuesto por motivos de rendimiento.

Otra aproximación por *hardware* fue publicada por Hyun-Chul Shin et al. en 2001 [139], que asocia pesos a cada texel en función de un cálculo del área cubierta por el pixel en ese texel.

En ese mismo año, Marc Olano, Shrijeet Mukherjee y Angus Dorbie propusieron una alternativa que permite realizar el filtrado anisotrópico en *hardware* que no dispone de esta característica o aumentar el grado de filtrado anisotrópico en el *hardware* que sí disponga de ella [120]. Para ello utiliza varios accesos a la estructura de *mipmap*. Los cálculos de los parámetros que determinan el área cubierta por el pixel en espacio textura se realizan por vértice (y por tanto con una frecuencia inferior a la que resultaría si se calculase por *fragment*), lo cual contribuye a reducir la cantidad de cálculos necesarios.

2.6.2. La textura de terreno global

Uno de los trabajos pioneros acerca del texturizado de terrenos fue publicado en 1994 por Michael Cosman, ingeniero de Evans & Shutherland. En su artículo "*Global Terrain Texture: Lowering the Cost*" [55], Cosman plantea la necesidad de las texturas por dos motivos: en primer lugar como referencia óptica para calibrar visualmente la posición, orientación y velocidad y en segundo lugar para añadir realismo y correlación geográfica cuando la textura se obtiene a partir de las fuentes adecuadas y se aplica al terreno perfectamente georreferenciada.

Centrándonos en el segundo aspecto, las texturas geoespecíficas, según Cosman existen dos enfoques para resolver el problema:

- Utilizar texturas individuales formando un mosaico adecuado al modelo del terreno.
- Utilizar un *hardware* especial que proporcione un espacio de textura único, continuo y sin uniones visibles.

Como corresponde a un ingeniero de una compañía que construye generadores de imágenes, Cosman defiende las ventajas del uso de un *hardware* dedicado para el manejo de una textura global frente al uso de soluciones *software* basadas en mosaicos de texturas, realizando un estudio detallado y comparando ambas aproximaciones al problema.

Cosman enumera los siguientes objetivos para la **textura de terreno global**:

1. La textura debe ser correlacionable con la fuente en todo lugar y fotorreconocible en lugares especificados.
2. La textura debe ser consistente en contenido, actualidad, correlación con otros aspectos de la simulación, y en atributos fotográficos más intuitivos como color, intensidad y saturación.
3. La resolución de la textura debe cumplir los requerimientos de flujo óptico de la misión, que pueden variar a lo largo de la base de datos según las actividades a realizar.

4. No debe limitar la interacción del usuario con el entorno. El usuario debe tener absoluta libertad para operar en cualquier lugar, de cualquier manera y a cualquier velocidad que sean necesarias.

5. La textura no debe contener artefactos distractorios debidos al modo en que es adquirida, reconstruida, aplicada al terreno o *renderizada* por el *hardware* gráfico.

Un aspecto fundamental en el tratamiento de las texturas es la **coherencia espacial**. Cosman expone que los procesos para preparar, almacenar y *renderizar* la textura hacen uso de estas propiedades de coherencia espacial para optimizar la calidad visual y el rendimiento. Estas propiedades se conservan dentro de las texturas manejadas por el *hardware* diseñado específicamente para ello (y que están severamente limitadas en tamaño). Sin embargo, en el caso de fragmentar la textura global (que teóricamente es ilimitada en tamaño) en un mosaico de texturas, la coherencia espacial se rompe en las uniones entre los trozos de ese mosaico, lo cual incumple el 5º de los objetivos planteados, o en el mejor de los casos hace que el manejo de la textura sea costoso, puesto que se pierde la eficiencia de las técnicas implementadas en el *hardware*.

Uno de los problemas principales de los sistemas basados en mosaicos de texturas es la incapacidad de realizar de forma eficiente las operaciones de filtrado (descritas en la sección 2.6.1) entre texturas contiguas. Esto produce artefactos que hacen patentes las uniones entre dichas texturas. Además, en el caso de utilizar filtrado trilineal basado en *mipmaps*, esta rotura de la coherencia espacial en los bordes se acentúa en cuanto se reduce la textura, aún cuando el filtrado de los niveles del *mipmap* se haya realizado teniendo en cuenta los texels de las texturas vecinas.

Cosman declara que la textura global de terreno no se repite, cada texel tiene una posición geográfica y se dibuja en una única posición de la geometría. Conceptualmente, el espacio textura se debe considerar infinito, continuo y no repetido. Sin embargo, en algunos casos puede ser necesario considerar la textura del terreno como repetida, por ejemplo en caso de utilizar una imagen de todo el planeta en coordenadas geográficas, se puede considerar que el borde oeste continúa con los texels del borde este y viceversa como una textura repetida a lo largo de la coordenada s, pues de lo contrario se produciría una unión visible en la zona del terreno correspondiente a los límites de la imagen.

Otra de las ideas que sugiere Cosman es incluir un detalle fino de textura para operaciones como despegues o aterrizajes, donde la imagen geoespecífica no ofrece la suficiente resolución. Este detalle se puede conseguir a base de combinar texturas genéricas repetidas sobre la imagen geoespecífica del terreno. Un aspecto que recalca es que en estos casos el *hardware* necesita tener una resolución numérica considerable para poder trabajar a escalas tan diferentes (alto margen dinámico) sin errores de precisión en los cálculos.

El terreno posee ciertas características especiales frente a otros modelos geométricos, las cuales simplifican su tratamiento. Tanto el espacio de geometría como el de textura son visualmente continuos y no se solapan con ellos mismos (salvo raras excepciones). Esto facilita tareas como el cálculo de las coordenadas de textura. Por otra parte, la extrema coherencia espacial del terreno ayuda a delimitar fácilmente la zona que puede ver el espectador y la que presumiblemente visitará en breve, de cara al diseño de cachés y sistemas de paginado. Otro de los aspectos interesantes indicados por Cosman es que los LOD bajos de la textura corresponderán a zonas del terreno que se visualizan con ángulos cercanos a la horizontal, lo cual puede afectar a la elección de los algoritmos de filtrado de textura para esos LOD.

En resumen, las características que se exigen al terreno y en base a las cuales se comparan las aproximaciones del mosaico de texturas y *hardware* dedicado, son las siguientes: la aplicación de la textura no debe tener uniones apreciables visualmente y debe ser geográficamente precisa a lo largo de toda la base de datos y en todo el espectro del terreno y sus niveles de detalle. Además, la textura debe comportarse bien en cuanto a la calidad de la imagen y el *aliasing*.

Mosaicos de texturas

Utilizando la aproximación del mosaico de texturas, la ausencia de uniones visibles se consigue asegurándose de que los espacios de las diferentes texturas utilizadas estén alineados, sean coincidentes en los bordes y su tamaño sea una potencia entera de 2. Además, los mapas de textura deben comenzar, terminar y estar alineados con aristas de los polígonos, puesto que el motor gráfico no puede aplicar diferentes texturas a diferentes partes de un polígono[18].

En los bordes de las texturas, el filtrado bilineal necesita texels que corresponden a las texturas vecinas. En estos casos se puede simplemente limitar el direccionamiento de la textura de forma que no alcance la zona donde necesita los texels de las texturas vecinas. Sin embargo, esto no se puede considerar una solución, puesto que provoca que los bordes de las texturas individuales del mosaico sean visibles.

Una solución a estas uniones visibles de las texturas es añadir un borde adicional de un texel de grosor a cada textura. Esto aumenta las necesidades de almacenamiento y complica la gestión de las texturas. Las coordenadas de mapeado deben ser ajustadas para tener en cuenta este borde y de forma diferente en cada LOD. Estas tareas deben ser realizadas por *fragment*, lo cual planteaba la necesidad de *hardware* adicional. También aumenta la complejidad de las texturas con zonas a diferentes resoluciones (es habitual la presencia de *insets* de alta resolución en zonas de singular interés).

[18] En la actualidad esto sí se puede realizar mediante un *fragment shader*, aunque no sin un coste considerable en rendimiento y uso de recursos de texturizado

Actualmente, la gestión de los bordes de la textura que plantea Cosman está soportada en el *hardware* gráfico, al menos desde la versión 1.1 de OpenGL, publicada en 1992. Sin embargo, la fuerte restricción de hacer coincidir exactamente los bordes de la textura con los bordes de los polígonos es inevitable y muy problemática.

La coincidencia en los bordes de la geometría mapeada por una misma textura con los de dicha textura, obliga a que los conjuntos de polígonos que forman el terreno se agrupen en bloques rectangulares o cuadrados, aunque interiormente puedan estar teselados de forma irregular. Además, esto afecta al tamaño máximo de los polígonos, que nunca pueden exceder la zona cubierta por un único mapa de textura. Además de complicar la gestión del modelo geométrico del terreno, se limita drásticamente su simplificación en las zonas llanas. En el caso de sistemas de gestión dinámica de los LOD de la geometría del terreno, la situación se hace más grave aún.

Otro problema grave es la combinación de los LOD de geometría y textura. Ambos LOD se manejan mediante mecanismos diferentes. Mientras que la textura se gestiona en función de la relación entre el tamaño del *fragment* proyectado en el espacio textura, los LOD de la geometría se determinan en función de las irregularidades del terreno y/o de la distancia a la cámara. Niveles finos de textura pueden estar aplicados a niveles poco detallados de geometría y viceversa, según la posición y orientación de la cámara respecto al terreno y las características de éste.

Por último, todo el proceso de desarrollo de las bases de datos debe adaptarse para tener en cuenta las restricciones impuestas por el mosaico de texturas. Los cambios en las bases de datos, tanto de textura como de geometría, frecuentemente necesitan una reconstrucción completa de ambas, lo cual puede ser un proceso costoso en tiempo y espacio de almacenamiento.

Hardware dedicado

La alternativa propuesta por Cosman al uso de mosaicos de texturas es el uso de *hardware* dedicado para el mapeado de texturas. Este *hardware*, implementado en los generadores de imágenes de Evans & Sutherland ESIG 4000 y ESIG 3000, permite gestionar la textura de terreno global como un único mapa de tamaño ilimitado. Esto garantiza la continuidad y evita la rotura de la coherencia espacial a lo largo de la textura, solucionando los problemas planteados en la aproximación mediante mosaicos de texturas.

No es necesario incluir la información de mapeado en la geometría, puesto que las coordenadas de textura son calculadas automáticamente por el *hardware* para cada vértice.

Se dispone de la capacidad de aplicar texturas adicionales de alto nivel de detalle, que correspondan a materiales (texturas genéricas), según la geología o el tipo de suelo, de forma adecuada a las necesidades de cada aplicación.

La memoria del sistema gráfico se organiza de forma que se mantienen

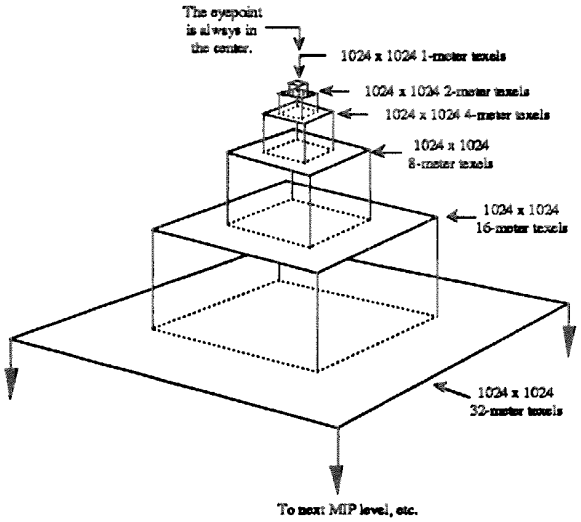


Figura 2.30: Organización de la memoria en la textura de terreno global de Cosman.

residentes sólo las porciones de la textura global que sean necesarias para cada situación. Esta memoria se organiza en LODs de textura que se paginan de manera independiente. Para cada nivel del *mipmap*, desde el más detallado, la cantidad de memoria utilizada será la misma, de forma que cada nivel tendrá texels de mayor tamaño, de hecho cubrirá cuatro veces el área geográfica cubierta por el nivel anterior a la mitad de resolución en cada eje (figura 2.30). Avanzando por los niveles, se llegará a un nivel que cubra toda el área visible para la aplicación en cuestión. Los niveles siguientes del *mipmap* se pueden almacenar todos en un único bloque de memoria del tamaño de los anteriores.

Durante la visualización, se gestiona la paginación de los niveles de textura para mantener una jerarquía de niveles centrada en el punto hacia el que mira el espectador.

El uso de memoria de textura y el tiempo de actualización (y por tanto el rendimiento del sistema) se pueden controlar para adaptarlos a las necesidades concretas de cada aplicación. Esto se realiza mediante un parámetro de configuración que determina el tamaño de la zona de memoria dedicada a cada nivel. La implementación original ofrecía dos valores posibles: regiones cuadradas de 512 y 1024 texels de lado. La zona de la textura que debe estar presente en cada nivel se determina a partir del centro de interés (posición de la cámara).

Los LOD inferiores (de menor detalle) tienen preferencia en la actualización. De esta forma, cuando se desborda el ancho de banda disponible para la actualización, simplemente se texturiza el terreno con una resolución inferior, pero con la textura correcta. Como esta situación suele producirse en situaciones donde la cámara se mueve a gran velocidad y se encuentra próxima a

la superficie, este efecto es perfectamente aceptable, puesto que el visualizar la textura más borrosa se corresponde con el comportamiento natural del ojo cuando capta imágenes en movimiento. Según la velocidad decrezca, el detalle disponible irá aumentando hasta alcanzar el máximo cuando la cámara se mueva a velocidades reducidas o se mantenga estática, que es cuando realmente el usuario podrá centrar su atención en los pequeños detalles del terreno.

Este sistema de texturizado no tiene ninguna relación con la poligonalización del terreno. Tanto la paginación como la selección de LODs de textura se realiza de forma completamente independiente de los de la geometría. Los modelos poligonales del terreno y las texturas se podrán desarrollar de forma independiente y en paralelo. La correlación vendrá dada únicamente por un *datum* geodésico común y las transformaciones necesarias para unificar los diferentes espacios de coordenadas. Por lo tanto, la estrategia de texturizado es topológicamente y topográficamente independiente del modelo geométrico o las estrategias de gestión de dicho modelo. Por lo tanto, no se condiciona en absoluto la elección del sistema de gestión de geometría y se facilita el cambio entre unos sistemas y otros, que resultará transparente al motor de texturizado. La generación de nuevas bases de datos o la modificación de las existentes también se facilita en el sentido de que se minimiza su impacto en el resto del sistema.

Las herramientas para el desarrollo de las bases de datos de textura de terreno global asumen un espacio de textura continuo e infinito. Los datos se referencian en coordenadas geográficas (latitud y longitud). No se limita la libertad de la creación de texturas a partir de imágenes heterogéneas en cuanto a particionamiento, tamaños, origen de los datos, proyecciones, etc. Simplemente, estas imágenes serán procesadas para constituir la textura de terreno global en el espacio de coordenadas que esta utilice.

Finalmente, Cosman pone de manifiesto la dificultad en conseguir buena información de textura de terreno. Mencionando algunos de los numerosos problemas y dificultades que plantea la obtención de estos datos, llega a concluir que los generadores de imágenes son capaces de presentar más y mejor textura de la que habitualmente los usuarios puedan llegar a conseguir.

Como solución a esta limitación en la disponibilidad de imagen del terreno de calidad y con una alta resolución, plantea la generación dinámica de texturas de alta resolución bajo demanda, a partir de información de bases de datos temáticas del terreno, como puede ser DFAD (*Digital Feature Analysis Data*). Estas texturas las define como *"fotorrealistas aunque no fotoespecíficas"*. Se aplican en base a la información temática del terreno, que consiste en entidades vectoriales (puntos, líneas o áreas) con una clasificación en características de la superficie terrestre que incluyen tanto elementos naturales como artificiales (lagos, ríos, bosques, glaciares, puentes, carreteras, líneas de ferrocarril, construcciones, áreas urbanas, etc).

2.6.3. *Clipmaps*

En verano de 1998, tres ingenieros de Silicon Graphics, Christopher C. Tanner, Christopher J. Migdal y Michael T. Jones, presentaron en el SIGGRAPH una técnica de mapeado de texturas arbitrariamente grandes en su artículo "*The Clipmap: A Virtual Mipmap*"[144].

Esta técnica ha alcanzado una gran popularidad, llegando a establecerse como la principal referencia en el mapeado de grandes texturas de terreno, a pesar de que conceptualmente no realiza una gran aportación sobre la propuesta cuatro años antes por Cosman [55] y, al igual que ésta, requiere *hardware* específico, disponible únicamente en las estaciones de gama alta de SGI con *hardware* gráfico Infinite Reality [114].

A grandes rasgos, se puede decir que esta técnica proporciona la capacidad de utilizar texturas arbitrariamente grandes con una ocupación limitada de memoria manteniendo una velocidad de *render* adecuada para aplicaciones críticas como simuladores de vuelo, esto es al menos 60 fotogramas por segundo. Tanner et al. plantean seis objetivos para esta técnica:

1. Debe soportar el *render* a plena velocidad utilizando un pequeño subconjunto de una textura arbitrariamente grande.

2. Debe ser posible actualizar rápidamente este subconjunto de la textura simultáneamente con el *render* en tiempo real.

3. No se debe forzar la subdivisión o cualquier otra restricción sobre la geometría texturizada.

4. El control de carga debe ser robusto y automático para evitar discontinuidades visuales en las situaciones de sobrecarga.

5. Debe ser posible integrar fácilmente esta técnica en las aplicaciones existentes.

6. El incremento del coste de la implementación de la técnica sobre el *hardware* existente debe ser reducido.

Aunque los objetivos son diferentes a los planteados por Cosman, más concretos y más centrados en el rendimiento que en la calidad, los medios para cumplirlos son similares en ambos casos. Tanner et al. también descartan la aproximación por mosaicos de texturas pero describen con mayor profundidad el diseño del sistema y añaden algunas aportaciones sobre el trabajo de Cosman.

La implementación de *clipmap* de SGI combina *hardware* de bajo nivel (implementado en la arquitectura InfiniteReality[114]) con un sistema *software* de alto nivel (implementado en IRIS Performer[127], y posteriormente

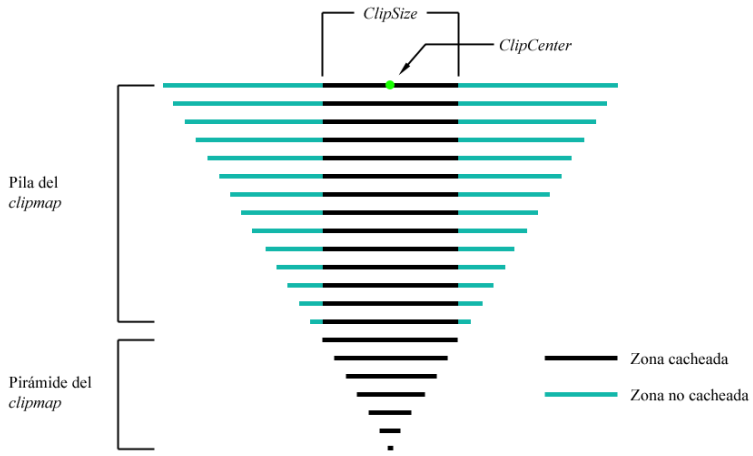


Figura 2.31: Estructura de un *clipmap*.

OpenGL Performer) que proporciona una interfaz de programación de aplicaciones para facilitar el uso de esta técnica tanto en aplicaciones nuevas como en las ya existentes.

El concepto de *clipmap* está basado en el de *mipmap* planteado por Williams [148], una pirámide de LODs de textura prefiltrados y almacenados en memoria. Si se analiza el uso de *mipmaps* de gran tamaño, se puede ver claramente que la mayoría de la pirámide no se usa en el *render* de una vista concreta, independientemente de la geometría texturizada. La base de esta técnica es el hecho de que el punto de vista y la resolución de la pantalla determinan el uso de la pirámide, y por lo tanto se propone la construcción de *hardware* que *renderize* a partir del subconjunto mínimo del *mipmap* necesario en cada fotograma.

El *clipmap* es una estructura actualizable de un *mipmap* parcial (ver figura 2.31), donde cada LOD ha sido recortado a un tamaño máximo. Este tamaño se determina mediante el parámetro denominado **ClipSize** (tamaño de recorte). La estructura resultante tiene forma de obelisco en lugar de pirámide, y se diferencian dos grupos de niveles: los niveles completos en memoria por tener tamaño de lado inferior o igual al ClipSize (que se denomina **pirámide del *clipmap***) y los niveles incompletos en memoria por tener un tamaño superior (que se denominan **pila del *clipmap***). El ClipSize debería ser mayor o igual que la resolución de pantalla, para garantizar que se pueda cubrir la totalidad del área visible con el nivel adecuado.

Los niveles de la pirámide del *clipmap* son exactamente iguales a los niveles del *mipmap* y están permanentemente en la memoria de texturas. Cada nivel de la pila del *clipmap* consiste en una ventana de tamaño ClipSize^2 texels del nivel correspondiente del *mipmap*, que está presente en memoria de texturas.

Otro concepto importante es el el centro de detalle o centro de interés, que se denomina **ClipCenter**. Las ventanas de los niveles del *mipmap* cacheadas por los niveles de la pila del *clipmap* estarán dispuestas en torno a este punto central. Cada nivel de la pila tendrá su ClipCenter, aunque lo habitual es que se establezca para el nivel 0 (máximo detalle) y se calcule en los niveles subsiguientes siguiendo la línea entre el ClipCenter del nivel 0 y el ápice de la pirámide. Esto garantiza que las zonas cacheadas por los niveles estarán dispuestas de forma concéntrica, teniendo el mayor detalle en torno al ClipCenter y bajando progresivamente al alejarse del mismo.

La base teórica del *clipmapping* es que con un ClipSize adecuado y una buena elección del ClipCenter, la información disponible en memoria de texturas es un superconjunto de la necesaria para *renderizar* con el máximo detalle. De ahí que la elección y la gestión de estos parámetros sean críticas.

La gran ventaja del *clipmap* frente al *mipmap* es el reducido uso de memoria. Para una textura de tamaño $2^n \times 2^n$, el uso de memoria (en texels) utilizando un *clipmap* con un ClipSize 2^m está indicado en la ecuación 2.9. En la ecuación 2.10 se muestra el uso de memoria de la misma textura con un *mipmap*. Lo más importante a destacar es que el uso de memoria de un *mipmap* crece exponencialmente con el tamaño de la textura, mientras que en el caso del *clipmap* este crecimiento es lineal.

$$4^m \left(n - m + \frac{4}{3} \right) \tag{2.9}$$

$$4^n \frac{4}{3} \tag{2.10}$$

La memoria utilizada por el *clipmap* tiene el tamaño suficiente para almacenar los texels necesarios para *renderizar* correctamente un fotograma. A medida que se desplaza el punto de vista, es necesario actualizar los contenidos de esta caché. Este proceso de actualización se realizará típicamente antes de *renderizar* cada fotograma. Se determina qué regiones hacen falta según la nueva posición del ClipCenter y se cargan en memoria. Debido a la elevada coherencia entre fotogramas en el contenido de la caché, los texels cargados se podrán reutilizar en siguientes fotogramas.

Para facilitar una reutilización de los texels cargados en la caché, la técnica de *clipmap* se basa en un direccionamiento toroidal de los niveles de la pila. Este direccionamiento toroidal evita tener que hacer costosos desplazamientos de los texels en memoria, de forma que simplemente hace falta cargar los nuevos texels que entran en la zona de interés sobre las áreas de memoria que ocupaban aquellos que abandonan la zona de interés (ver figura 2.32).

La cantidad de información que se necesita actualizar en un nivel es cuatro veces superior al siguiente nivel (de menor detalle), de forma que la cantidad de memoria que se necesita paginar en los niveles inferiores será muy baja.

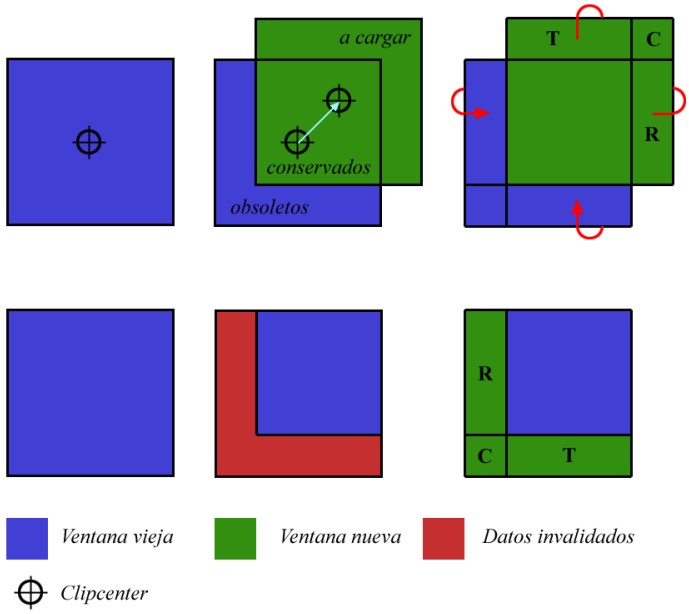


Figura 2.32: Direccionamiento toroidal.

Un aspecto importante de esta técnica es el ancho de banda de carga de texturas que necesita para mantener la caché completamente actualizada de forma que se proporcione el máximo detalle de textura. Este requerimiento está determinado por tres parámetros: el tamaño de la pila del *clipmap* (que a su vez depende del tamaño total de la textura "virtual"), el ClipSize y la velocidad a la que se desplaza el ClipCenter. La cota superior al tiempo de actualización de la caché es el necesario para recargar la pila del *clipmap* al completo (lo cual es independiente de la velocidad de navegación).

El *render* utilizando *clipmaps* se realiza de forma similar al uso de *mipmaps*, con ciertos aspectos adicionales a tener en cuenta. Además de calcular el LOD adecuado para un *fragment*, exactamente igual que se hace al aplicar un *mipmap*, se debe tener en cuenta cuál es el mayor LOD disponible en la caché del *clipmap* para la zona correspondiente a ese *fragment*. Por otra parte, a la hora de direccionar los texels del nivel o los niveles del *clipmap* que se vayan a aplicar, se debe tener en cuenta el direccionamiento toroidal de dichos niveles, aplicando el desplazamiento correspondiente para que cada texel quede ubicado en la posición correcta.

Por limitaciones del *hardware* en cuanto a precisión numérica, la implementación de *clipmap* de SGI no podía direccionar simultáneamente más de 16 niveles[19].

[19]Como referencia se plantea el objetivo de una textura de todo el planeta a 1 m/texel de resolución. Siendo el radio ecuatorial de la tierra 40.000 km aproximadamente, esta

La solución planteada para superar este límite del *hardware* consiste en lo que denominaron "virtualizar los *clipmaps*". Esto consiste en utilizar los 16 niveles disponibles en el *hardware* para cachear un subconjunto de los niveles del *clipmap* (virtual) completo. En un mismo fotograma se pueden utilizar todos los niveles del *clipmap* virtual siempre que se *renderizen* diferentes partes de la geometría estableciendo la ventana de niveles adecuada para mapear esa porción del terreno. Tanner et al. afirman que mientras se cumplan las condiciones de que cada polígono de la escena no se extienda en los niveles de la textura más allá de 16 niveles y antes de *renderizar* cada polígono se establezca la ventana de 16 niveles adecuada a dicho polígono, se puede utilizar cualquier tamaño de pirámide (siempre que la caché correspondiente quepa en la memoria de texturas disponible).

Para optimizar la transferencia de textura desde el almacenamiento persistente hasta la memoria de texturas, se utiliza una caché de segundo nivel en memoria principal. Se utilizan varios procesos o hilos de ejecución en paralelo para gestionar este flujo de información, al menos uno encargado de la planificación del flujo de información, otro encargado de transferir los datos desde disco a memoria principal y otro para actualizar de forma incremental la caché de los niveles del *clipmap* en la memoria de texturas del sistema gráfico.

La caché de segundo nivel en memoria principal se organiza en bloques cuadrados cuyo tamaño se establece para maximizar la tasa de transferencia en las lecturas desde disco. Estos bloques se gestionan mediante un direccionamiento toroidal similar al utilizado en la caché en TRAM. La caché realiza también una precarga de los bloques de las zonas próximas que rodean a la que está en la pila del *clipmap* y que presumiblemente serán solicitadas en un futuro inmediato.

En casos de sobrecarga de la caché de segundo nivel, debidos a desplazamientos rápidos, el sistema debe gestionar las prioridades de los bloques solicitados a dicha caché. Como ya planteaba Cosman, la información se carga por orden de LOD, dando mayor prioridad a los de menor detalle (y mayor extensión geográfica). Pero además, dentro de cada nivel se debe tener en cuenta qué bloques tienen mayor o menor importancia, para evitar desbordar el sistema encolando peticiones de carga que correspondan a zonas que ya se han dejado atrás. En esta situación se deben descartar directamente esas peticiones.

El posicionamiento del ClipCenter es un aspecto crítico y muy dependiente de la aplicación, por lo que se delega esa responsabilidad en el usuario. Algunas aplicaciones tienen un campo de visión reducido y la cámara se dirige hacia lugares distantes, en cuyos casos el ClipCenter se debe situar en la zona hacia la que mira dicha cámara. En otros casos, cuando es habitual que se gire la vista con rapidez y los desplazamientos son cercanos al

textura supone una pirámide de 26 niveles.

suelo, es más práctico situar el ClipCenter bajo la posición de la cámara, de forma que en esos giros no se desborde el ancho de banda disponible para la actualización del *clipmap*. Un ejemplo típico de este último caso son las aplicaciones de realidad virtual donde se captura la posición de la cabeza del usuario, que tiene libertad para girar inmediatamente en cualquier dirección.

Es necesario un control del tiempo dedicado a la actualización de la caché para evitar que se desborde el tiempo de *render* disponible en cada fotograma. Por este motivo, en el arranque se construye una tabla de tiempos de carga para diferentes tamaños de bloques de textura. Esta tabla se utiliza durante la ejecución para realizar estimaciones de los tiempos de actualización de los niveles en cada fotograma. De esta forma, se actualizan los diferentes niveles partiendo de los de menor detalle y subiendo hasta que el cálculo estimado indique que no queda tiempo suficiente para actualizar un nivel. El nivel máximo disponible se ajusta en consecuencia y se termina la fase de actualización de ese fotograma.

La implementación de *clipmap* de SGI permite la presencia de zonas de interés aisladas, con una resolución muy alta, contenidas dentro de una textura que cubre un área mayor y con una resolución menor. Tanner et al. denominan a estas islas como ***high-resolution insets***. Aplicaciones típicas de esto son los aeropuertos y zonas circundantes en un simulador de vuelo, o la inclusión de imagen aérea de alta resolución de las ciudades más importantes dentro de un área geográfica más extensa con imagen de satélite a una resolución menor.

Por las características de diseño del sistema, los *insets* deben disponer de un borde de al menos ClipSize texels alrededor. En caso contrario, no se podría aplicar la información del inset hasta que la ventana cacheada de esos niveles superiores estuviese contenida completamente dentro de la zona donde hay información, provocando cambios bruscos en el nivel de detalle visible.

2.6.4. MP-Grid

En noviembre de 1998, Tobias Hüttner de la Universidad de Tübingen, publicó otra técnica de manejo de texturas de alta resolución [92]. Esta técnica, utilizada en sistemas como FlyAway[94], divide la imagen de gran resolución en una serie de fragmentos rectangulares de igual tamaño. Cada uno de estos fragmentos constituye una textura OpenGL con su pirámide *mipmap*[148] que se precalcula y se almacena en disco. Estos fragmentos no tienen por qué ser cuadrados, aunque ambas dimensiones deben ser potencia de 2. La imagen global se escala a un tamaño que sea divisible en fragmentos potencia de dos, aunque el tamaño de esta imagen en sí no tiene por qué serlo. Esta es una de las ventajas que esgrimen frente a otras técnicas como los *clipmaps*[144].

La colección de pirámides de *mipmap* de todos los fragmentos de la ima-

gen global se denominan **MP-Grid** (*MIPmap pyramid grid*). Cada una de las pirámides del MP-Grid se actualiza en función del punto de vista para cargar aquellos niveles de *mipmap* necesarios en cada caso. El cálculo de los niveles necesarios en cada pirámide, se realiza a partir de las cajas envolventes (*bounding box*) de la geometría que se texturiza.

En este sentido, se trata de un sistema más versátil que *clipmapping*, puesto que no se limita a una zona estrictamente cuadrada alrededor del centro de detalle. El aprovechamiento de la memoria puede ser más eficiente, teniendo en cuenta factores como oclusión, pendiente de cada zona del terreno, etc.

Como ventajas adicionales, el autor indica la posibilidad de compartir la caché entre varios visores, lo cual no es posible en *clipmapping*, y el no necesitar establecer un centro de detalle, lo que plantea como una tarea crítica y bastante compleja.

Sin embargo, la gestión de la caché de estos niveles es más compleja y problemática que en *clipmapping*. La flexibilidad de MP-Grid se paga con problemas de fragmentación de memoria. El autor no indica cuál es el mecanismo para la gestión de la memoria de texturas.

El mayor de los problemas de esta técnica es que al usar directamente una textura OpenGL para cada fragmento, la geometría se debe teselar para que todos los polígonos estén contenidos dentro de una única textura. Esto es un proceso lento, costoso y sobre todo absolutamente intrusivo para el motor de gestión de la geometría, que debe modificar su diseño e implementación para cumplir con esta grave restricción. Hüttner propone el almacenar los resultados de la geometría teselada para coincidir con las texturas en *display lists*, a razón de una *display list* por cada textura, para acelerar el *render*. Sin embargo esto sólo es posible suponiendo que la geometría sea estática, lo cual es muy poco habitual hoy en día, puesto que casi todas las técnicas actuales utilizan LODs continuos.

Hüttner propuso unas extensiones al *hardware* gráfico para posibilitar la implementación eficiente de MP-Grid, de cuya implementación ho ha habido noticias hasta la fecha.

2.6.5. Otros sistemas de texturizado

Rabinovich y Gotsman (1997)

Rabinovich y Gotsman publicaron en 1997 una técnica de gestión de terrenos en entornos con recursos limitados [123], centrada en su funcionamiento en una red de baja velocidad.

Para la gestión de la textura, ésta se almacena en el servidor dividiendo la extensión del terreno en un mosaico de trozos. Estos trozos de textura están comprimidos utilizando una técnica de wavelets progresivos, que reduce al 30 % aproximadamente las necesidades de almacenamiento y transmisión. El

cliente solicita al servidor los trozos que necesita y los carga en la textura situada en la memoria gráfica, que sufre el límite de tamaño impuesto por el *hardware*.

Para optimizar el aprovechamiento del ancho de banda de la red, sólo se transmiten los trozos de textura que están contenidos en la zona visualizada y a la resolución adecuada. Estos trozos se reconstruyen en la memoria de texturas a partir de las wavelets hasta el nivel de resolución necesario. Cuanto más bajo sea este nivel, menos información será necesario transmitir por la red desde el servidor.

No se considera la situación de anisotropía, la resolución utilizada de la textura se decide únicamente en función de la distancia de la cámara al punto donde se aplica.

Otro problema grave de esta técnica es que ocasionalmente tiene que desplazar los contenidos de la textura debido al movimiento del punto de vista.

Cline y Egbert (1998)

David Cline y Parris K. Egbert publicaron en 1998 un artículo sobre el uso de texturas de gran tamaño [52]. En él describen una arquitectura *software* para la gestión de texturas genéricas, no orientadas exclusivamente a terrenos.

Los sistemas tradicionales de texturizado para aplicaciones interactivas consideran la memoria de texturas como un recurso finito y las texturas como recursos atómicos. Esto produce un límite al tamaño de las texturas que se pueden utilizar en una escena y un retardo inicial durante la carga de todas las texturas. Como alternativa, plantean el considerar una textura como un recurso limitado por ancho de banda en lugar de un recurso finito.

Los objetivos que se plantean son los siguientes:

1. Se deben soportar texturas de gran tamaño, entendiendo como tales aquellas que superan la capacidad de la memoria de texturas e incluso de la memoria principal.

2. El tiempo de arranque de la escena debe ser lo más rápido posible e independiente de las texturas que se utilicen.

3. El sistema debe poder implementarse en estaciones gráficas de gama baja utilizando las APIs gráficas actuales.

4. La diferencia en fotogramas por segundo entre usar el sistema de caché de texturas propuesto y utilizar texturas estáticas que quepan en la memoria de texturas debe ser inapreciable.

5. Se debe dar prioridad a mantener una buena respuesta interactiva antes que presentar un elevado detalle instantáneo

6. El sistema debe comportarse razonablemente bien accediendo a las texturas a través de una red de baja velocidad.

7. El sistema debería proporcionar un marco de trabajo genérico para una amplia variedad de aplicaciones que requieran texturas de gran tamaño.

Centrándose en el problema de aumentar el tamaño disponible en las texturas para utilizar en aplicaciones interactivas en tiempo real, destacan dos tipos de técnicas para resolverlo: las técnicas de compresión de texturas y el uso de cachés. Puesto que aún con el uso de compresión, la memoria disponible se desborda facilmente, el sistema propuesto se centra en el estudio de la segunda de estas alternativas.

Cline y Egbert alaban las virtudes de los *clipmaps*, especialmente el considerar una textura única en lugar de trocear la imagen en pequeñas texturas imponiendo severas restricciones a la división de la geometría para que coincida con los límites de esas texturas. Se descartan únicamente por el importante requerimiento de *hardware* específico, lo cual incumple el tercer objetivo planteado.

Proponen una caché a tres niveles, siendo la memoria de textura el primer nivel, la memoria principal el segundo y el disco el tercer nivel. Esto añade un nivel de caché al planteamiento de *clipmapping*, puesto que se considera el acceso remoto a los datos servidos por red.

Identifican cuatro cuellos de botella que limitan el rendimiento de una aplicación interactiva respecto al uso de texturas:

- Transferencias de disco o de la red. Esto produce el bloqueo de la aplicación hasta que los datos son recibidos completamente.

- Transformación de las texturas. Aquí consideran procesos como la descompresión, reescalado para convertir a tamaños que sean potencias enteras de 2, generación de *mipmaps*, etc.

- Desbordamiento de la memoria de texturas. Cuando esto sucede, las texturas se transfieren entre la memoria de texturas y la memoria principal, lo cual provoca una caída drástica del rendimiento.

- Paginado de la memoria virtual. De la misma forma que la memoria de texturas, cuando se desborda la memoria principal, se produce un paginado a disco que hace caer el rendimiento del sistema.

Este artículo presenta muy bien la problemática de estos cuellos de botella, el uso de servidores de textura, y el flujo de la textura a lo largo de todo el sistema. Va más allá de un simple sistema de gestión de memoria virtual, realizando el control del ancho de banda disponible en las diferentes partes del sistema para cumplir los exigentes requisitos de un sistema de

visualización en tiempo real. Se trata en profundidad el funcionamiento de la jerarquía de las cachés, gestión de las peticiones, prioridades, etc.

Las texturas están almacenadas en el origen de datos con todos los niveles de *mipmap* pregenerados y troceados. Los fragmentos se organizan mediante una estructura *quadtree* [130].

Los niveles de menor resolución están disponibles de forma casi inmediata y el detalle se va refinando progresivamente allí donde se necesita. A la hora de *renderizar* una primitiva geométrica, se determina cuál es el nivel adecuado (dentro de los disponibles) para esos polígonos y se aplican como una textura normal.

Esto impone una restricción muy grave sobre la división de la geometría, común a la práctica totalidad de los sistemas que gestionan la imagen global como un mosaico de texturas, que se ilustra con un sencillo ejemplo en la figura 2.33.

El cuadrado grande de la figura representa la imagen global de la textura. Sobre ella están representadas las divisiones de los dos primeros niveles del *quadtree*, de forma que la textura A corresponde al nivel de menor detalle que cubre toda la extensión, la B a la cuarta parte de esta y la C la octava parte. Los polígonos numerados como 1, 2 y 3 representan tres casos de mapeado.

El polígono 1 se podrá mapear con la textura C, aplicando el máximo detalle disponible. El polígono 2, aún siendo del mismo tamaño y muy inferior al área cubierta por las texturas de mayor resolución, el detalle más alto que puede conseguir es el de la textura B, puesto que ninguna textura del siguiente nivel (de mayor resolución) lo cubre completamente. El caso más grave se produce en el polígono 3. No importa la resolución de que se disponga en esta textura, por mucha profundidad que tenga el *quadtree*, el polígono 3 sólo se puede mapear con la textura que cubre el área completa o alguna de sus versiones reducidas.

La única forma de solucionar este problema es la división de los polígonos 2 y 3 para que sus límites coincidan exactamente con el borde de las texturas. Además de dividir los polígonos, es necesario que estén en primitivas diferentes, puesto que se les van a aplicar distintas texturas. Esto, además de aumentar el número de polígonos necesarios, dificulta su organización en estructuras eficientes que los combinen en primitivas como *tristrips* o *trifans*[135].

Otra limitación muy importante es que este sistema, a diferencia de *clipmapping* no realiza filtrado trilineal para reducir el *aliasing*. Se podría modificar fácilmente para incluir esta característica sustituyendo cada textura por una pirámide de *mipmap*, pero esto incrementaría la ocupación de memoria y el ancho de banda necesario para las transferencias.

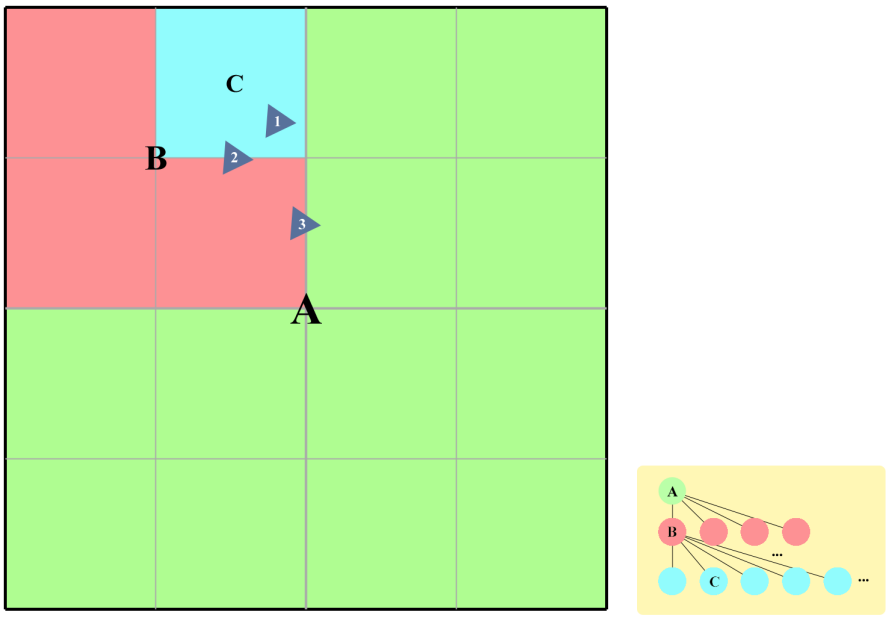


Figura 2.33: Problema de la organización en *quadtrees* de los mosaicos de texturas.

Döllner et. al (2000)

En el año 2000, Jürgen Döllner et al. publicaron un trabajo sobre técnicas de texturizado para la visualización de terreno [65].

Al igual que otras técnicas, provoca un fuerte acoplamiento entre geometría y textura. A partir de la textura original se construye una pirámide y un árbol de fragmentos de textura a diferentes niveles de detalle, donde cada fragmento de textura está asociado a un fragmento de geometría del modelo geométrico multirresolución. Sin embargo, en el artículo se sugieren ideas muy interesantes.

Las texturas no tienen por qué limitarse a simples imágenes que se superponen a la geometría, sino que se pueden utilizar mapas de componentes de iluminación (reflexión difusa o especular), mapas temáticos, mapas de visibilidad (canal alfa), mapas topográficos con el aspecto de la geometría del terreno, preiluminación, mapas de normales, texturas generadas dinámicamente, etc.

Plantean combinar varias capas de texturas multirresolución (donde cada capa se compone de su pirámide y su árbol) a través de diferentes técnicas, como fundidos (*blending*) y máscaras (*masking*). Para ello realizan varios pases de *render*, que pueden reducirse en caso de disponer de *hardware* con capacidad de multitextura. Una de las posibles aplicaciones de esta combinación de múltiples capas es el uso de texturas animadas para visualizar datos espacio-temporales.

Un interesante ejemplo de aplicación de la combinación de capas es lo que

denominan ***texture lenses***, que se pueden utilizar por ejemplo para desviar la atención del usuario hacia una zona concreta del terreno.

Klein y Schilling (2002)

La técnica para *render* de modelos multirresolución progresivos de terreno, publicada por Reinhard Klein y Andreas Schilling en 2002 [98], incluye la gestión de textura de muy alta resolución aplicada a terrenos extensos.

En este artículo, respecto al mapeado de textura, se mencionan las técnicas de *clipmapping* y MP-Grid.

Respecto a la primera, además de la evidente necesidad de *hardware* específico, se resalta la problemática de la ubicación del ClipCenter para un aprovechamiento óptimo de la memoria de texturas. Aún estableciendo un ClipCenter independiente para cada LOD de la textura, la forma cuadrada de las regiones cacheadas, no se adapta de manera óptima a la forma triangular del terreno contenido en el *frustum* de la cámara. Por otra parte, no se pueden aprovechar las técnicas de filtrado por oclusión (*oclussion culling*) para reducir la textura que se necesita cachear.

MP-Grid resuelve las limitaciones descritas de los *clipmaps*, pudiendo adaptarse los LODs de textura de forma más flexible, pero introduce otros problemas diferentes. El principal problema es la limitación de la estructura de la geometría a las texturas utilizadas, que por una parte evita la simplificación del terreno y por otra impide el correcto filtrado de las texturas cuando el tamaño en espacio pantalla de las mismas es inferior a un *fragment*. Además, en estos casos, la cantidad de *mipmaps* necesarios la hace inviable. Estos problemas se agravan, por tanto, cuanto más se aleja la cámara del terreno, cubriendo los *fragments* una mayor extensión geográfica.

La técnica presentada combina MP-Grid con la organización en *quadtree* del terreno, y tiene diversas limitaciones por estar basada en un mosaico de texturas. La precisión de la geometría que se *renderizará* está limitado por la resolución de la textura en esa zona. Por otra parte, el único criterio utilizado para la elección de la textura es la distancia a la cámara, no se tiene en cuenta el espacio pantalla.

BDAM/P-BDAM, Cignoni et al. (2003)

En los trabajos de Cignoni et al. (BDAM [49] y P-BDAM [50]), se afirma que pocas técnicas desacoplan totalmente la gestión de los LOD de textura y geometría. La única técnica en su conocimiento que lo haga completamente es *clipmapping*, que requiere *hardware* especial. Acerca de la limitación de adaptación de los límites de la geometría a las texturas que forman el mosaico de la imagen global, resaltan que provocan mayor dificultad en las técnicas basadas en TIN.

Estas técnicas, BDAM y P-BDAM, gestionan la textura del terreno troceándola en porciones y organizándola mediante una estructura jerárquica de tipo *quadtree*. Como la geometría se organiza en una estructura de tipo *bintree*, se asocia cada nodo (textura) del *quadtree* a dos nodos adyacentes (mallas geométricas) del *bintree*.

Durante el *render*, se realiza un recorrido descendente del *quadtree* de texturas hasta localizar un nivel con un error aceptable en espacio pantalla. Se aplica esa textura y se comienza un refinamiento de los dos *bintrees* correspondientes a esa textura hasta alcanzar un nivel en el cual el error geométrico en espacio pantalla es aceptable. De esta forma se minimizan los cambios de estado al aplicar una única vez cada textura.

La imagen de textura se procesa generando todos los niveles de detalle mediante un filtrado progresivo y opcionalmente se comprime utilizando el formato DXT1. Se utiliza el borde de la textura para preservar la continuidad.

En el caso de P-BDAM, los autores hacen referencia a ciertos problemas inherentes a un sistema de escala planetaria. En primer lugar los límites de la precisión de los sistemas gráficos actuales, que trabajan con aritmética flotante de precisión simple (32 bits).

Por otra parte, el tiempo necesario para preprocesar (*off-line*) las bases de datos se dispara a esta escala, convirtiéndose en un importante aspecto a tener en cuenta. Por este motivo plantean la necesidad de soluciones escalables que realicen los procesos en paralelo.

P-BDAM organiza la superficie del planeta particionándolo en parcelas cuadradas, que resultan en una colección de jerarquías BDAM. El número y el tamaño de las divisiones depende del tamaño de los datos originales. Se asegura que un número en coma flotante de precisión simple tenga la precisión suficiente para representar las coordenadas locales y que el tamaño de las estructuras de datos multirresolución utilizadas no supere los límites del sistema operativo. Esto limita a 2^{23} texels en cada dirección[20] y a al límite máximo de bloque de memoria contigua que se puede mapear (típicamente menos de 3 GB).

Esta división en bloques se utiliza únicamente para solucionar los problemas de precisión de la aritmética flotante, no afectando a la continuidad en las uniones entre bloques.

Geometry Clipmaps, Losasso y Hoppe (2004)

La técnica de Geometry Clipmaps [107] publicada por Losasso y Hoppe en 2004 asocia a cada nivel del *clipmap* de geometría una textura, de forma que la gestión de LODs se realiza conjuntamente. Descartan el uso de *mipmapping* por *hardware* porque, además de incrementar un 33 % la ocupación de memoria, puede producir unas uniones visibles en los cambios de

[20]Se limita a 2^{23} texels porque el formato de aritmética flotante en precisión simple (ANSI/IEEE Standard 754-1985) asigna 23 bits para la mantisa.

nivel. Por este motivo realizan la gestión de LODs de textura completamente sincronizados con los de geometría, incluyendo la zona de transición. Por lo tanto, los LOD de textura se basan en un criterio de distancia a la cámara, sin tener en cuenta los parámetros habituales en el texturizado, basados en las derivadas en espacio pantalla.

Holkner (2004)

Alex Holkner, en su variante de Geometry Clipmaps basada en GPUs programables [89], cita el uso de las texturas asociadas al *clipmap* de geometría como mapas de iluminación para mejorar el aspecto de relieve del terreno. Estos mapas de iluminación se generan a partir de la posición del sol y se filtran para obtener los diferentes LOD utilizando un filtro sinc-Blackman al igual que el utilizado para la geometría del terreno. Utilizan un *fragment shader* para realizar un filtrado bilineal de la textura con un filtrado adicional en las zonas de transición, necesario para evitar uniones visibles entre LODs. Holkner sugiere el uso de una resolución para el mapa de iluminación que sea el doble de la utilizada para el mapa de elevación.

Brodersen (2005)

En el año 2005, Anders Brodersen publicó un artículo sobre visualización de grandes extensiones de terreno texturizado [44]. En él, tras un repaso por el estado del arte de las técnicas de visualización de terreno, indica que en todas existe un fuerte acoplamiento entre los algoritmos de gestión de LODs de geometría y textura con la única excepción de los *clipmaps*, que requieren *hardware* especial.

La técnica de Brodersen se basa en la de De Boer [61] para la gestión de la geometría y acopla esta con las texturas que forman un mosaico al igual que en la mayoría de las técnicas anteriores.

Virtual Textures, Ephanov y Coleman (2006)

Anton Ephanov y Chris Coleman, de la compañía MultiGen-Paradigm, presentaron el año 2006 una técnica de gestión de texturas de gran tamaño, que denominaron *Virtual Textures* [68].

En primer lugar, introducen las dos opciones habituales para la gestión de texturas de gran tamaño: los mosaicos de texturas y los *clipmaps*, mencionando las virtudes y limitaciones de cada una de ellas. Acerca de los *clipmaps* indican la necesidad de *hardware* específico y la imposibilidad de combinar con otras texturas y técnicas de texturizado basadas en *shaders*.

La técnica que denominan *Virtual Textures* (VT), se sitúa a medio camino entre una y otra aproximación. Como requisitos de diseño plantearon los siguientes:

1. Las VT deben comportarse de forma tan similar como sea posible a una textura normal. Se aplicarán a las primitivas utilizando coordenadas de textura igual que se realizaría con cualquier otra textura.

2. Deben poder utilizarse desde técnicas basadas en *vertex* y *fragment shaders*.

3. Debe ser posible combinar varias VTs en una única pasada utilizando las capacidades multitextura de las GPUs modernas.

4. La implementación no debe necesitar más que las capacidades disponibles en OpenGL 1.2.

5. Se deben cumplir los requerimientos de las aplicaciones de visualización en tiempo real de alto rendimiento: *frame rate* estable a un mínimo de 60 fps, *render* en una única pasada y sostener el tráfico de datos a velocidades de vuelo Mach 1 o mayores.

6. Deben ser lo suficientemente flexibles para aceptar datos almacenados o servidos por *streaming* en diferentes formatos y tamaños de fichero.

Esta técnica sigue la filosofía de los *clipmap* en cuanto a cachear un subconjunto del *mipmap* de la imagen completa. Para cada nivel del *mipmap* se almacena un subconjunto de esa información, centrado en un punto de interés, que se denomina nivel de VT. Cada nivel de la VT está compuesto por una matriz cuadrada de $n \times n$ trozos (*tiles*) que corresponden a texturas normales de OpenGL, las cuales poseen su propio *mipmap*. Los niveles se numeran igual que en OpenGL, siendo el nivel 0 el de mayor resolución.

Estas texturas con *mipmap* que componen los niveles de la VT, se gestionan con una técnica de doble *buffer*. Realmente existen dos texturas para cada *tile* de un nivel de la VT: una utilizada para la actualización de datos y otra para la visualización. Cuando se termina el proceso de actualización, ambas texturas intercambian los papeles y se comienza un nuevo ciclo.

La posición del centro de detalle se realiza de forma independiente en cada nivel.

La técnica descrita por Ephanov y Coleman no considera en absoluto cómo se almacenan los datos de la textura, asumen que hay un mecanismo para obtener un área determinada de un nivel de resolución determinado.

Muestran diversos ejemplos de aplicación de las VT, aparte de la inmediata de texturas de color, como pueden ser mapas de rugosidad o mapas de normales para calcular iluminación por pixel, simulación de sensores, mapas de elevación, mapas de horizonte[21] o mapas de iluminación nocturnos para simular de forma eficiente las fuentes de luz existentes durante la noche

[21] Los mapas de horizonte almacenan ángulos que sirven para calcular si un punto del terreno queda iluminado por una dirección de luz concreta. Se utilizan habitualmente para calcular la proyección de sombras arrojadas por fuentes de luz como el sol o la luna.

(ciudades, carreteras, etc.). También indican la posibilidad de combinar las VT con otras técnicas para añadir detalle adicional, como la extensión *detail texture* de SGI.

En muchos casos, estas aplicaciones especiales tienen una problemática adicional. Por ejemplo los mápas temáticos, cuya información consiste generalmente en índices, o los mapas de normales, no se pueden filtrar con las técnicas habituales, como los *mipmaps*, ni utilizar técnicas de compresión de imagen diseñadas para información de color.

El funcionamiento de la técnica de VT consiste principalmente en un algoritmo para la transformación de las coordenadas de textura, que convierte las coordenadas de la textura global en las coordenadas específicas para un *tile* concreto de un nivel de resolución. Esta transformación consiste en una traslación y un escalado, por lo que se puede aplicar mediante la matriz de transformación de textura de OpenGL, sin necesidad de utilizar *shaders* ni características adicionales a las disponibles en OpenGL 1.2. En la ecuación 2.11 se detalla la transformación a realizar sobre las coordenadas de textura (u, v) en el espacio de la textura global para obtener las coordenadas (U, V) en el espacio del *tile* centrado en las coordenadas $(u_{ctr}, v_{ctr}$ perteneciente al nivel *level*, con un tamaño de fragmento 2^{tile} texels y un tamaño de la VT completa de 2^{vt} texels.

$$\begin{aligned} U &= 2^{vt-level-tile} \times u + (0{,}5 - 2^{vt-level-tile} \times u_{ctr}) \\ V &= 2^{vt-level-tile} \times v + (0{,}5 - 2^{vt-level-tile} \times v_{ctr}) \end{aligned} \tag{2.11}$$

Tras la transformación de las coordenadas de textura, la siguiente fase es la aplicación de estas texturas a las primitivas geométricas. Para esto, se utiliza un algoritmo de selección de *tiles* de nivel a los objetos geométricos. Cada uno de estos objetos tiene una caja envolvente alineada con los ejes (*axis aligned bounding box* o AABB) definida en el espacio de coordenadas de textura de la VT. A partir de la AABB de cada elemento geométrico, se realiza la selección del nivel de textura y el *tile* adecuado para mapearlo.

Cada nivel de la VT tiene también una AABB asociada, que consiste en la unión de todas las AABB de sus *tiles*. Para cada uno de estos *tiles*, la AABB estará centrada en el *tile* y su tamaño se define en la ecuación 2.12, donde la extensión en texels de la VT es dim_{vt} y la extensión en texels del *tile* del nivel *level* es dim_{level}.

$$\begin{aligned} u_{min} &= (u_{ctr} - 0{,}5 \times dim_{level} \times 2^{level})/dim_{vt} \\ u_{max} &= (u_{ctr} + 0{,}5 \times dim_{level} \times 2^{level})/dim_{vt} \\ v_{min} &= (v_{ctr} - 0{,}5 \times dim_{level} \times 2^{level})/dim_{vt} \\ v_{max} &= (v_{ctr} + 0{,}5 \times dim_{level} \times 2^{level})/dim_{vt} \end{aligned} \tag{2.12}$$

Para cada elemento geométrico, se itera por la jerarquía de AABBs de los niveles y *tiles* de la VT, comparándolas con la AABB de la geometría, hasta localizar el *tile* que la cubre completamente, puesto que esa será la

textura OpenGL que se podrá aplicar a esa geometría. Finalmente se realiza una ordenación por textura de los elementos geométricos a *renderizar*, para minimizar el número de cambios de estado en la máquina OpenGL (en concreto la activación de texturas), de forma que se optimice el rendimiento en el *render*.

Las gran limitación de este diseño basado en el *pipeline* fijo de OpenGL es que para poder aplicar una textura, ésta debe cubrir completamente a la geometría texturizada. Se puede dar el caso incluso de que en un nivel haya disponible información suficiente para cubrir la geometría, pero no con un único *tile*, en cuyo caso se deberá ir a niveles de menor resolución hasta conseguir cubrirla. Este caso es equivalente al ilustrado en la figura 2.33.

Para solucionar esta limitación, Ephanov y Coleman proponen un diseño alternativo, que incumple el cuarto punto de los requisitos que plantearon inicialmente, utilizando *shaders* y multitextura. De esta forma pueden combinar en un *fragment shader* varias texturas que cubran la geometría. Se comienza por las de mayor detalle y se van aplicando por este orden, enmascarando las texturas en las que no hay información disponible para ese *fragment* y utilizando las de menos detalle sólo donde no existe información en las de más detalle.

La ventaja de esta alternativa es que no hace falta teselar la geometría para alcanzar mayores niveles de detalle. La desventaja es que, aparte de necesitar capacidad de programación en la GPU, se utilizan más unidades de textura, lo cual dificulta los objetivos planteados inicialmente, especialmente la combinación de varias VTs o VTs con otras texturas y otros *shaders*. Además, el número de unidades disponibles en la GPU suele ser limitado y el *fragment shader* puede tener un impacto importante sobre el rendimiento del *render*.

Aunque el uso de *shaders* puede reducir el problema, el limitado número de unidades de textura disponibles hace necesario un cierto nivel de teselado de la geometría para alcanzar una resolución adecuada. Al fin y al cabo, sigue tratándose de una técnica basada en un mosaico de imágenes, con los problemas subyacentes a todas estas técnicas, como ya se ha mencionado repetidamente a lo largo de este capítulo.

Clipmaps independientes del *hardware*, Seoane et al. (2007)

En el 2007 fue publicada por Seoane et al. una técnica que aproxima la funcionalidad de los *clipmaps* (con algunas ventajas adicionales) a los sistemas gráficos más modestos, sin necesidad de GPUs programables [137]. Sigue muy fielmente la filosofía de *clipmapping* adaptándola para su implementación en cualquier sistema sin más requisitos que OpenGL 1.2.

Las principales ventajas de esta técnica son la facilidad de tratamiento independiente de las bases de datos de geometría y textura, el uso de una única textura OpenGL, lo cual posibilita la combinación de varios *clipmaps*

utilizando la capacidad de multitextura de la GPU o utilizando la programabilidad mediante *shaders* si se dispone de ella.

También se permite tener un nivel de detalle heterogéneo a lo largo de la extensión de la textura, con áreas de muy alta resolución en los lugares de especial interés o importancia.

Se puede realizar un filtrado trilineal mediante *mipmaps* o anisotrópico mediante la extensión `EXT_texture_filter_anisotropic` en caso de estar disponible en el *hardware* gráfico. De esta forma se aprovechan las capacidades del *hardware* gráfico para conseguir una gran calidad y un rendimiento muy alto.

A diferencia de las VT de Ephanov [68], esta técnica sí realiza una actualización toroidal de los niveles, de forma similar, aunque no exactamente igual, a los *clipmaps* originales [144]. Además, la actualización de los niveles de resolución es progresiva en anillos concéntricos comenzando desde el centro de detalle, de forma que se puede aplicar la zona disponible de un nivel de detalle que aún no ha sido completamente actualizado.

La principal limitación es que el tamaño de los bloques de geometría que se *renderize* determina el máximo nivel que pueden alcanzar. Es importante remarcar que es el tamaño y no la organización o la posición de las divisiones de la geometría, como sucedía en la gran mayoría de las otras técnicas (ver figura 2.33). Esto facilita su uso con mallas irregulares (TIN).

Otra diferencia importante respecto a otras técnicas basadas en mosaicos de texturas, como MP-Grid[92], es que no se limita la simplificación de geometría, se puede reducir el espacio global hasta un polígono conservando un mapeado correcto, perfectamente filtrado con filtro trilineal o anisotrópico.

El consumo de memoria es fijo, conocido a priori y configurable. No se producen problemas de fragmentación de memoria, y la sencillez en el esquema de actualización y *render* proporciona un resultado muy eficiente con una calidad excelente.

Otras implementaciones de *clipmaps*

Con las capacidades de programación y el rendimiento de las nuevas generaciones de GPUs, están comenzando a surgir algunas implementaciones de *clipmaps* con la capacidad de texturizado por *fragment* del original disponible en SGI, como la desarrollada por nVidia utilizando arrays de texturas sobre DirectX 10 [118].

Como parte del trabajo de esta tesis doctoral, se ha realizado una solución completa basada en la experiencia de [137] y perfectamente integrada dentro de una arquitectura genérica que permite combinar diferentes *clipmaps* y texturas normales junto con cualquier *shader*, que será descrita en detalle en los próximos capítulos. Permite aprovechar todos los texels disponibles en el *clipmap* independientemente de la geometría texturizada, realiza filtrado anisotrópico y necesita una única unidad de texturizado, dejando libres el

resto para combinar con otras texturas o efectos.

2.7. Datos vectoriales 2D sobre el terreno

La información vectorial manejada en los SIG, tal y como se ha descrito en el apartado 2.2, está representada generalmente sobre el plano, sin información explícita de altura. Para visualizarla sobre el modelo digital 3D del terreno, los datos vectoriales se deben adaptar perfectamente a la superficie de dicho terreno.

Existen varias aproximaciones para resolver la problemática de la adaptación de los datos vectoriales 2D sobre el terreno 3D. Principalmente se pueden dividir en dos estrategias independientes: construir la geometría 3D correspondiente a los datos vectoriales procedentes de los SIG o convertir esos datos en una textura que se mapea sobre el terreno.

2.7.1. Construcción de geometría 3D

Una de las posibles aproximaciones para la representación de información vectorial 2D sobre el terreno 3D es la construcción de geometría 3D correspondiente a esa información vectorial 2D.

En el caso de los puntos, simplemente se toma la altura del modelo del terreno correspondiente a su posición en 2D. La representación se puede realizar mediante modelos 3D sencillos como esferas para indicar la posición del elemento, modelos más complejos como iconos 3D, textos con información relativa al elemento que se sitúa o cualquier combinación de los anteriores. Es habitual también colocar la información 2D, ya sean textos o imágenes, de forma que esté siempre orientada hacia la cámara (*billboards*) para facilitar su legibilidad, y en una posición elevada sobre la superficie del terreno para evitar que quede parcialmente enterrado. En estos casos es útil colocar una línea a modo de poste para ubicar la posición exacta del elemento sobre el suelo. En la figura 2.34 se ilustran varios ejemplos de representación de información SIG puntual.

El segundo tipo de información vectorial indicado en la sección 2.2 son las líneas. En este caso la situación se complica notablemente más que en el caso de la información puntual. La primera aproximación consiste en colocar los vértices de las líneas en la altura correspondiente del terreno, es decir, proyectar verticalmente sobre el terreno los vértices de las líneas. Sin embargo, salvo casos excepcionales, esta aproximación producirá errores debidos a la irregularidad del terreno (ver figura 2.35).

Para solucionar este problema, es imprescindible fragmentar las líneas, introduciendo nuevos vértices en las aristas de los polígonos del terreno que crucen, para que coincidan exactamente con la superficie del terreno (ver figura 2.36).

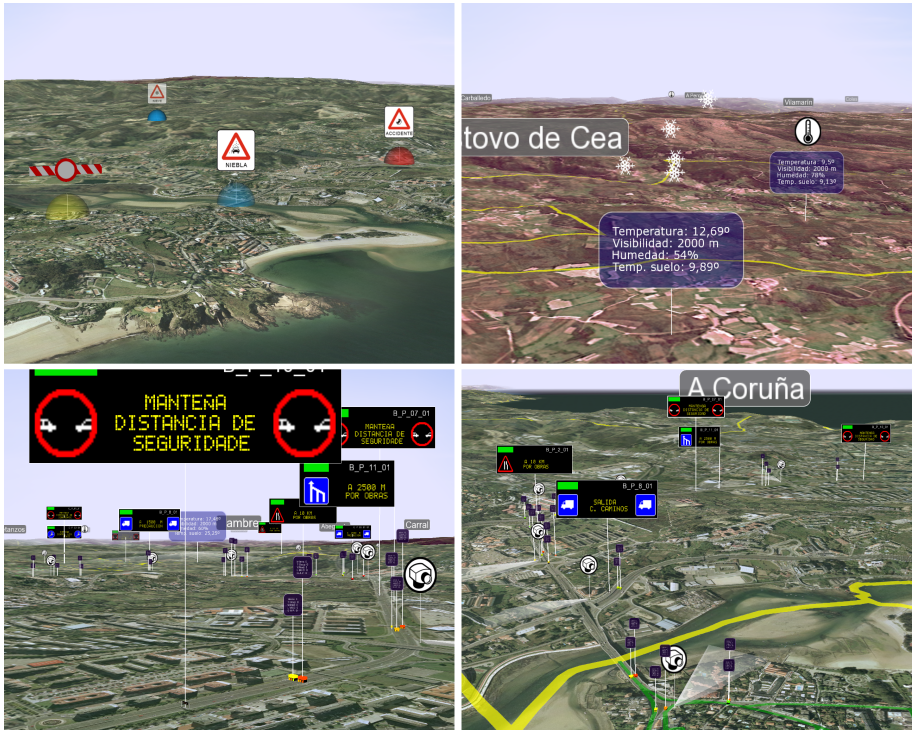


Figura 2.34: Ejemplos de representación de información SIG puntual.

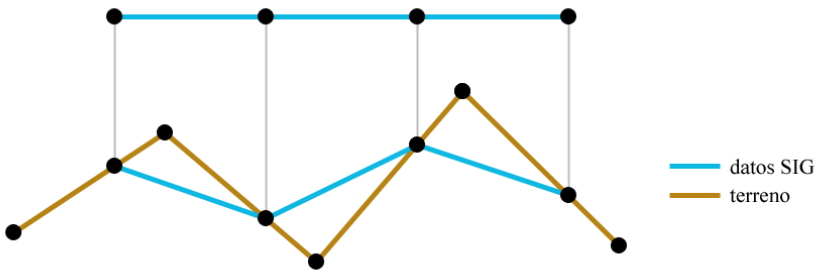


Figura 2.35: Problemas en la adaptación de líneas 2D sobre el terreno 3D.

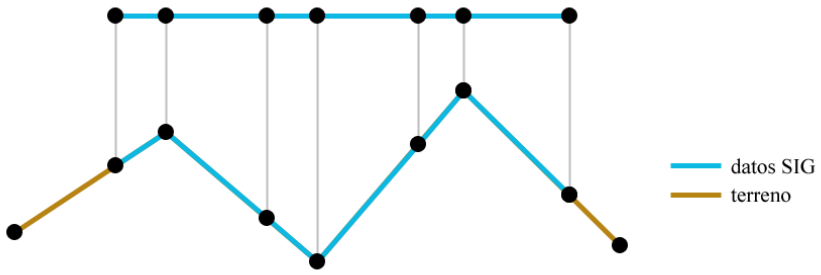


Figura 2.36: Adaptación de líneas 2D sobre el terreno 3D.

Figura 2.37: Problemas en la visualización de polígonos coplanares. a) Efecto *Z-fighting* debido a errores de precisión en el *Z-buffer* b) Visualización correcta

En el caso de información de áreas, que se construye a base de polígonos rellenos, el problema es similar al de las líneas, pero más acentuado en dos sentidos: los errores son más graves visualmente y la complejidad del teselado de dichos polígonos es mayor que en el caso de las líneas. Esto se traduce en un mayor coste de cálculo.

Uno de los problemas de dibujar la geometría de la información vectorial sobre la geometría del terreno es que debido a errores de precisión numérica en el algoritmo *Z-buffer*, se producen efectos visualmente muy graves. Este fenómeno se conoce habitualmente como lucha en Z (*Z-fighting*), cosido (*stitching*) o sangrado (*bleeding*), y su aspecto se ilustra en la figura 2.37a.

La solución a este efecto indeseado puede ser elevar la geometría correspondiente a la información SIG una cierta distancia sobre el terreno. El desplazamiento aplicado debe escogerse con cuidado. La información geográfica parecerá estar flotando sobre el terreno si el desplazamiento es excesivo y se producirán errores de precisión en el *Z-buffer* (*Z-fighting*) si es insuficiente. En caso de variar en un margen amplio la altura de vuelo, es imprescindible una gestión dinámica de este desplazamiento, como realiza Google Earth

[30].

Por supuesto, la profundidad de pixel asignada al *Z-buffer* determina absolutamente la gravedad del problema de *Z-fighting*.

Existe un mecanismo de OpenGL[135] que puede servir de ayuda a este problema. La llamada `glPolygonOffset()` permite establecer un desplazamiento que se añade al valor de profundidad (en espacio pantalla) calculado en todos los *fragment* generados en el proceso de *rasterización* de los polígonos. El desplazamiento aplicado tiene en cuenta dos parámetros: la pendiente del polígono *rasterizado* y la resolución del *frame buffer*.

Aunque se trata de una herramienta de gran utilidad, no es infalible. Si bien es cierto que se puede evitar completamente el efecto de que la información vectorial flote sobre el terreno, como contrapartida se pueden producir errores de ordenación en profundidad (Z), por ejemplo que una línea que debiera estar oculta tras una montaña aparezca sobre esta por aplicársele un desplazamiento excesivo en Z. Por lo tanto, se mantiene la necesidad de un ajuste dinámico de estos desplazamientos en profundidad en los casos en que la distancia a la cámara sea muy variable. Es importante indicar también que la experiencia nos ha mostrado que el ajuste de los parámetros de la función `glPolygonOffset()` es bastante delicado, se necesita un ajuste muy fino y desviaciones mínimas producen errores visibles.

Otro problema adicional, en el caso habitual de utilizar LOD geométricos para el terreno, es que esta rotura de las líneas y los polígonos se debe realizar de forma dinámica y adaptarse continuamente para permanecer perfectamente sincronizados con la superficie del terreno durante la visualización. Esto incrementa todavía más la complejidad de gestionar la información vectorial 2D aplicada perfectamente sobre el terreno, especialmente en el caso de los polígonos.

Pero más allá del coste de estos cálculos asociados a la adaptación de la geometría de la información SIG vectorial a la geometría del terreno, existen otros problemas. Según cuál sea el diseño del sistema de gestión de LODs geométricos, puede ser muy difícil (o incluso imposible) acceder a la información necesaria para modificar las primitivas de la información SIG vectorial adaptándolas a la superficie del terreno.

Aún en el caso de solventar esta dificultad, la solución será completamente dependiente de la técnica de gestión de los LODs de geometría de terreno y no se podrá utilizar con técnicas distintas.

Un último punto a indicar es el problema de *aliasing* de geometría, que será muy visible en los bordes de los polígonos y a lo largo de las líneas, sobre todo cuando se alejen de la cámara. Para evitar este efecto, se deben utilizar técnicas de *antialiasing* de geometría, que básicamente consistirán en mecanismos de *multisampling* o *supersampling* [34] implementados en el *hardware* gráfico, con un coste de tiempo de cálculo considerable.

Wartell et al. (2003)

Zachary Wartell et al., del Graphics, Visualization and Usability Center (GVU) del Georgia Institute of Technology, publicaron en el año 2003 un artículo [146] donde describen cómo añaden la capacidad de visualizar información vectorial al sistema VGIS [99, 102]. La técnica descrita se basa en la adaptación de la geometría vectorial 2D del SIG a la superficie 3D del terreno.

Asocian el concepto de LOD en gráficos por computador con el de simplificación geométrica en los SIG. La simplificación geométrica es un caso especial del concepto de generalización en la cartografía: métodos para representar un determinado elemento o conjunto de elementos geográficos con diferentes aspectos según la escala a la que se visualicen.

El sistema VGIS maneja conjuntos de datos cuyo volumen desborda la capacidad de la memoria del sistema. Por lo tanto realiza paginado dinámico de los datos, tanto de la elevación del terreno como de la imagen del terreno y de la información SIG vectorial.

El gestionar la información SIG vectorial como geometría genera una serie de dificultades importantes. Puesto que utilizan el algoritmo de LODs continuos de Lindstrom para la gestión de la geometría del terreno [103], los triángulos que forman el terreno varían en cada fotograma, y por tanto se necesita que la geometría de los datos vectoriales se actualice también en cada fotograma para adaptarse a la superficie de dicho terreno.

Describen una estructura de datos (tc-DAG por *triangle clipping directed acyclic graph*) y los algoritmos asociados, optimizados para el *render* en tiempo real. Estos algoritmos están adaptados al algoritmo de Lindstrom [103], siendo necesaria su modificación para utilizarlos con técnicas diferentes.

Los autores realizan una comparación de las técnicas basadas en geometría y las basadas en textura para la aplicación de la información SIG vectorial. Al uso de textura achacan los siguientes problemas:

- La aproximación trivial de *rasterizar* la información vectorial al máximo detalle de la imagen de terreno para mezclarla con ésta, provoca que al alejarse la cámara del terreno se filtren las líneas hasta desaparecer.

- Por el contrario, al acercarse al terreno más allá de la resolución disponible en la imagen, los datos vectoriales se observarán "pixelados", cuando, por la propia naturaleza vectorial de dichos datos, deberían ser independientes de la distancia de visión y de la resolución disponible en la imagen del terreno.

- Como conclusión de los dos puntos anteriores, los autores achacan a estas técnicas el limitar la resolución de la información vectorial a la resolución disponible en la imagen de la textura.

- Las técnicas que *rasterizan* la información vectorial para generar texturas, según Wartell et al., no disponen de la flexibilidad para variar su aspecto de forma interactiva. En concreto hacen referencia a la capacidad de activar o desactivar conjuntos de información, ajustar parámetros como el grosor de las líneas, color, estilo de línea (discontinua, punteada, etc.) con una respuesta interactiva.

Consideran la posibilidad de que una aproximación basada en textura, solucionando los problemas anteriores, gestione la resolución de la información vectorial de forma independiente de la de la imagen del terreno (*adaptive polyline-as-texture solution*). Este tipo de solución trabajaría con ambos conjuntos de información en texturas diferentes con resoluciones independientes, y necesitaría realizar una generación dinámica de la textura con los vectores *renderizados*. Indican también la necesidad de un sistema de caché con políticas de invalidación de los datos y la simplificación dinámica de la geometría *rasterizada* según el nivel de detalle.

Un aspecto a remarcar del trabajo de Wartell et al. es que hace referencia únicamente a polilíneas, en ningún caso a polígonos. La gestión de polígonos aumentaría todavía más la complejidad del algoritmo y su acoplamiento con el sistema de gestión de la geometría del terreno.

Agrawal et al. (2006)

Otra técnica posterior basada en geometría fue propuesta en 2006 por Agrawal et al. [33]. Está basada en un sistema de visualización que divide la geometría y la textura en un mosaico de trozos cuadrados de igual tamaño. Se mantienen en memoria nueve de estos trozos, centrados en la posición del espectador.

En muchos sentidos, esta técnica es similar a la anterior de Wartell et al. [146], pero adaptada a otro sistema de visualización de terreno, que nuevamente acopla geometría y textura. También se limita a información vectorial compuesta por polilíneas.

En este caso, los autores presentan algunos datos acerca del rendimiento del sistema. Indican el número de triángulos y los fotogramas por segundo que obtienen en diferentes situaciones.

Schilling et al. (2007)

Uno de los trabajos recientes en este campo es el publicado en marzo de 2007 por Schilling et al. [132].

Los autores proponen un sistema de representación de la información SIG vectorial basado en geometría. Este sistema está pensado para funcionar a través de la *web* y con dispositivos móviles, por lo que se trata de un sistema ligero que se basa en los estándares del OGC[15].

Por estos motivos no trata de forma dinámica el terreno, no utiliza LODs ni *streaming* de geometría. Las escenas servidas a los clientes son estáticas.

Descartan el uso de una aproximación basada en texturas en favor del uso de geometría por los siguientes motivos:

- Los navegadores *web* estándar no soportan técnicas de *render* a textura.
- La cantidad de datos transmitidos debe reducirse tanto como sea posible, especialmente considerando las redes móviles de baja velocidad.
- Las imágenes *raster* son ineficientes para representar datos poligonales.

Para eliminar los problemas de los polígonos coplanares y el *Z-buffer*, típicos de las aproximaciones basadas en geometría, tal y como ya se ha descrito, esta técnica integra la información SIG vectorial dentro de la propia superficie del terreno. El terreno, formado por una TIN, se retesela mediante una implementación del algoritmo de Delaunay restringido.

Como el resultado, que no altera la forma del terreno, es una única superficie continua que contiene las áreas correspondientes a la información SIG vectorial, se evitan los problemas del *Z-buffer* puesto que no existen polígonos coplanares. Los polígonos correspondientes a esta información vectorial se marcan como asociados a la capa SIG de la que provienen. Posteriormente se aplicará a esos polígonos el color y las características de aspecto correspondientes a la capa a la que pertenecen. Además, las aristas pertenecientes a la información SIG vectorial se marcan como "restringidas" para evitar que sean alteradas en procesos posteriores de la malla del terreno.

Finalmente, comparan la técnica propuesta con una alternativa basada en mapas de textura. Teniendo en cuenta sus objetivos concretos, el tamaño de las imágenes de textura aumentan la necesidad de ancho de banda en la comunicación además de ofrecer una calidad final peor. Tratándose de un modelo de terreno estático que se prepara off-line sin restricciones importantes de tiempo, la solución que proponen es la más adecuada para esa situación concreta.

2.7.2. Proyección mediante texturizado

La mayor ventaja del uso de texturas para representar la información vectorial sobre el terreno es que se desacopla la información vectorial de la geometría del terreno, permitiendo una gestión independiente de los LODs y no imponiendo ninguna restricción a la gestión de ambos conjuntos de datos.

Por otra parte, se eliminan por completo los problemas de la geometría anteriormente mencionados como los errores de precisión en el *Z-buffer*.

Los trabajos basados en geometría, descritos en el apartado anterior, achacan principalmente el hecho de que la calidad de visualización está limitada por la resolución y al acercarse al terreno las líneas o los bordes de los

polígonos se muestran dentados y/o borrosos. Sin embargo, en situaciones donde la cámara está más lejos, el aspecto puede ser notablemente mejor que con el uso de geometría, puesto que con un filtrado adecuado y el uso de niveles de detalle de textura se puede reducir el *aliasing* en las líneas y los bordes de los polígonos. Aunque la textura se genere dinámicamente, habitualmente, el coste de generación incluyendo las técnicas de *antialiasing* se amortiza entre varios fotogramas.

Kersting y Döllner (2002)

Una aproximación mediante texturas fue explorada por Oliver Kersting y Jürgen Döllner en un artículo [96] publicado en noviembre de 2002.

Tras enumerar las dos alternativas para solucionar el problema: uso de primitivas geométricas y uso de texturas con la información vectorial *rasterizada*, se decantan por la segunda debido a las dificultades de la primera para adaptarse a sistemas con LOD continuos, modificando esa geometría cada fotograma, y por los problemas derivados del algoritmo *Z-buffer*.

Descartan también la generación de la información *raster* mediante un preproceso off-line, principalmente por los requisitos en espacio de almacenamiento que eso supone y por que la resolución no se puede variar sin volver a realizar el preproceso. Además, este proceso es lento y costoso. Por otra parte, la única forma de combinar de forma interactiva diferentes conjuntos de información vectorial es utilizando multitextura. Sin embargo, esto limita a la combinación de un número reducido de capas, tanto por la limitación en las unidades de textura del *hardware* gráfico (o el impacto en el rendimiento de realizar múltiples pases de *render*) como por el espacio de almacenamiento que necesita esa información *rasterizada*.

Por todos estos motivos, proponen la gestión de la información vectorial mediante un *scene graph* que se *renderizará* bajo demanda sobre una textura que posteriormente se aplica al terreno. El hecho de generar los diferentes LODs de las imágenes de textura bajo demanda facilita la visualización de información dinámica en tiempo real y al mismo tiempo la adaptación de la representación de los datos vectoriales a las condiciones de visualización. Por ejemplo si la visual es cercana a la horizontal, los textos se pueden representar como carteles (*billboards*), mientras que en las vistas cenitales se pueden *rasterizar* sobre la propia textura en la orientación adecuada para que se lean correctamente. También se plantea la posibilidad de modificación interactiva de los datos vectoriales.

En la implementación descrita, utilizan *P-buffers* para el *render*. *Renderizan* mediante OpenGL utilizando el *P-buffer* como *frame buffer*. El contenido de este *P-buffer* se copia a una textura que finalmente se aplicará sobre la geometría del terreno. Sin embargo, existe acoplamiento entre geometría y textura, puesto que las texturas se gestionan como un mosaico de imágenes cuyas divisiones deben coincidir exactamente con los bloques de

geometría.

No se considera en ningún momento los problemas de *aliasing* de las texturas. Se utiliza un único LOD para las texturas generadas bajo demanda, por lo que el filtrado utilizado sólo podrá ser por proximidad o bilineal, por lo que es de esperar que se experimenten los problemas de *aliasing* descritos en la sección 2.6.1.

Los autores no ofrecen ningún dato sobre el rendimiento de la implementación, por lo que no se han podido evaluar adecuadamente los resultados.

Brooks y Whalley (2005)

Stephen Brooks y Jaqueline L. Whalley presentaron en 2005 un sistema híbrido 2D/3D para la visualización de información geográfica [45]. Se basan en la idea planteada por Springmeyer [143] de que las vistas en 2D se suelen utilizar para establecer relaciones precisas, mientras que las vistas en 3D ayudan a la adquisición de conocimiento cualitativo. Por lo tanto, ambas dimensionalidades tienen distintas ventajas. El trabajo de Brooks y Whalley combina ambas vistas perfectamente integradas en un mismo sistema para optimizar el rendimiento final.

Combinan un terreno 3D base con un sistema vertical de capas. El terreno 3D base a su vez se compone de al menos una capa espacial (topografía) y puede tener una o varias capas no espaciales (datos de visualización).

Para la topografía del terreno utilizan mallas regulares. Sobre esta geometría aplican texturas de satélite o bien texturas generadas sintéticamente a partir de los datos de elevación a las que se añade un cierto grado de ruido fractal para evitar un aspecto excesivamente limpio.

Las capas se pueden desplazar en vertical. En su posición más baja toman la forma del terreno 3D. Según se elevan hacia una determinada altura m, la elevación del terreno se reduce hasta convertirse en un plano donde se puede apreciar la capa como en las vistas 2D de un SIG tradicional pero colocada en perspectiva y alineada con el terreno 3D que tiene debajo. De esta forma se pueden examinar simultáneamente las dos vistas.

Las capas mostradas sobre el terreno 3D base están recortadas por cuatro planos de corte perpendiculares y equidistantes al centro del área visualizada.

Por encima de la altura m donde se convierte en un plano, la capa se hace progresivamente transparente, llegando a desaparecer por completo en la posición más alta de la pantalla. En todo momento permanece visible el borde de la capa con una línea del color asociado a esa capa (el mismo que se ve en la leyenda y en el manipulador que permite seleccionar y desplazar la capa).

El sistema propuesto por Brooks y Whalley trabaja únicamente capas *raster*. La información vectorial la precalculan sobre la textura de la capa correspondiente, y por lo tanto se limita a información estática.

No mencionan en absoluto cómo gestionan el mapeado de textura de sus capas. Tampoco mencionan nada acerca de la extensión de terreno ni el volumen de datos que pueden manejar, ni si implementan algún sistema de paginación (*out-of-core*). No incluyen datos acerca del rendimiento interactivo del sistema, especialmente en los desplazamientos entre diferentes zonas del terreno.

El sistema propuesto en esta tesis, dada su independencia de la geometría, podría aplicarse de forma transparente a un sistema como el de Brooks y Whalley, mejorando la calidad de la información *raster*, el rendimiento, la escalabilidad y posibilitando la inclusión de información dinámica.

2.7.3. Otras aproximaciones

Aunque las técnicas para representación de información SIG vectorial 2D sobre un modelo 3D del terreno se dividen principalmente en las dos estrategias descritas (uso de primitivas geométricas y uso de texturas), existen otras técnicas que no caen completamente dentro de ninguna de ellas, bien porque combinan parte de ambas o bien porque usan aproximaciones diferentes. A continuación se describen algunas de estas técnicas.

Schneider et al. (2005)

Martin Schneider et al. publicaron en 2005 una técnica para *renderizar* datos vectoriales complejos sobre modelos 3D de terreno [133] que combina la estrategia de uso de texturas con la de uso de geometría.

Indican las limitaciones de ambas aproximaciones, para finalmente proponer un sistema híbrido compuesto realmente de dos técnicas independientes. En cada caso se aplica una u otra según resulte más adecuado teniendo en cuenta las ventajas e inconvenientes ya descritos.

Se basan en un sistema de visualización [145] que estructura el terreno dividiéndolo en bloques de geometría y textura de igual tamaño organizados en un *quadtree*. Cada nodo del *quadtree* almacena una versión simplificada precalculada del terreno basada en una TIN.

El sistema basado en *rasterización* a textura bajo demanda utiliza una técnica de teselado eficiente de polígonos, propuesta por Guthe et al. [84]. Utiliza también una técnica de reparametrización de la textura para reducir el *aliasing* de perspectiva inspirada en el mapeado de sombras (*shadow mapping*). Para la creación de las texturas utilizan *P-buffers* o FBOs (*frame buffer objects*) en caso de que dicha extensión esté disponible.

El sistema de visualización de los datos vectoriales basado en geometría realiza un preproceso off-line donde se genera una colección de primitivas geométricas para cada LOD del *quadtree*. En tiempo de ejecución simplemente se aplica a cada nodo del *quadtree* la versión adecuada de la geometría 3D de los datos vectoriales 2D. Por lo tanto, en este caso se evitan los problemas

de *aliasing* de las texturas, especialmente cuando la cámara se acerca mucho al terreno, pero a cambio sólo se puede utilizar para información estática.

La aproximación basada en textura es más adecuada para elementos vectoriales poligonales de gran tamaño o que necesitan modificarse en tiempo real. En cambio la geometría resulta adecuada para elementos estáticos que necesitan una alta calidad de visualización o que tienen un tamaño reducido.

Schneider y Klein (2007)

Martin Schneider y Reinhard Klein presentaron en 2007 una técnica para *renderizar* datos vectoriales sobre el terreno 3D [134] basada en un uso ingenioso del *stencil buffer*, similar a una técnica propuesta en el libro rojo de OpenGL para el cierre de objetos sólidos seccionados por un plano de corte [43]. Esta técnica también se incluyó en el visualizador de terreno anteriormente mencionado [145].

Descartan el uso de texturas para dibujar la información vectorial sobre el terreno achacándole una precisión limitada por la resolución de la textura, que resulta en problemas de *aliasing*. Estos problemas pueden reducirse incrementando el uso de memoria de texturas, recurso muy limitado y valioso.

También critican las técnicas basadas en geometría por los problemas ya descritos, principalmente la necesidad de adaptar la geometría de los datos SIG vectoriales a los LODs de la geometría del terreno. Esto incrementa la cantidad de primitivas necesarias en caso de precalcularlas o el tiempo de cálculo en caso de generarlas en tiempo real. También resaltan el problema de la geometría coplanar y los consiguientes errores de precisión en el *Z-buffer*.

La técnica propuesta se basa en la extrusión de los datos vectoriales para formar poliedros con los que se genera una máscara en el *stencil buffer*. La máscara generada corresponde a la proyección de los datos vectoriales sobre la superficie del terreno en espacio pantalla.

La gran ventaja de este método es que trabaja en espacio pantalla pero tiene exactitud por pixel. Evita las limitaciones y los problemas de las dos aproximaciones clásicas mediante textura y geometría. No menos importante es el aspecto de que su forma de funcionamiento la hace absolutamente independiente del motor de geometría de terreno utilizado.

Sin embargo, también tiene ciertas limitaciones. Puesto que se basa en la generación de una máscara y trabaja en espacio objeto, sólo se puede aplicar un color sólido a los elementos vectoriales. Es imposible, por ejemplo, aplicar simbología específica a los elementos vectoriales (por ejemplo carreteras) o aplicarles textura con el mapeado correcto. Tampoco se puede, tal y como se plantea esta técnica, aplicar técnicas de iluminación o *shading* a los elementos vectoriales. Sería posible con ciertas modificaciones, pero haría falta *renderizar* la geometría del terreno en numerosas ocasiones, con la consiguiente sobrecarga en el *pipeline* gráfico.

En cualquier caso, es imprescindible realizar un pase de *render* por cada

objeto que tenga diferente color, lo cual puede llegar a afectar al rendimiento en caso de un número elevado de elementos diferentes.

Por otra parte, el incremento de las primitivas respecto a las aproximaciones por textura puede no ser muy grave para elementos sencillos y no muy numerosos, principalmente puntos, polilíneas o polígonos, pero sí puede convertirse en un cuello de botella *renderizando* otros elementos como textos.

En cualquier caso, los autores no presentan ningún dato acerca del rendimiento de esta técnica, lo cual hace imposible evaluar su eficiencia real en comparación con otras técnicas.

2.8. Limitaciones de los sistemas actuales

2.8.1. VGIS (*Virtual Geographic Information System*)

El sistema VGIS, desarrollado en el Georgia Institute of Technology, es uno de los pioneros en la integración de los SIG tradicionales con la visualización 3D interactiva. Está descrito en [99] y [102], y utiliza los trabajos de Lindstrom et al. [103] para la gestión de LODs continuos de geometría de terreno, Wartell et al. [146] para la visualización de polilíneas 2D sobre el terreno, Faust et al. [70] para la definición de un modelo global de terreno, Jiang et al. [149] para la gestión de información atmosférica y Davis et al. [60] para la gestión de volúmenes de información a través de cachés (*out-of-core rendering*) teniendo en cuenta la intención y percepción del usuario de manera específica para el terreno.

Este trabajo no profundiza demasiado en el sistema de gestión de texturas, que está acoplado al de geometría, manejado mediante el algoritmo de Lindstrom. Utiliza una organización jerárquica en *quadtrees* cuyos nodos corresponden a regiones cuadradas en coordenadas geográficas. Los datos *raster* asociados a los nodos del *quadtree* tienen tamaño cuadrado cuyo lado es una potencia entera de 2, y que cuadruplican su tamaño en cada nivel del árbol que se desciende[22], formando en esencia una estructura piramidal con los LOD precalculados y troceados en un mosaico de imágenes.

Para enmascarar zonas donde no hay información y para suavizar las uniones entre diferentes conjuntos de datos, utilizan un canal alfa tanto en las imágenes como en los datos de elevación.

La técnica de selección del LOD de textura es bastante pobre, debido a su acoplamiento con la geometría. No utiliza *mipmaps* ni filtrado anisotrópico, y por lo tanto, en muchas situaciones las texturas se verán excesivamente borrosas. Por otra parte, la gestión de las texturas es ineficiente, puesto que se crean y destruyen durante la visualización. Esto resulta costoso en tiempo de cálculo, complica el esquema de gestión de la caché de texturas en TRAM y puede producir problemas de fragmentación en la TRAM.

[22] Asumiendo que la raíz está en la parte más alta del árbol.

Aunque sin duda VGIS es un sistema muy completo, maduro, con una arquitectura muy flexible, versátil y potente, la gestión de las texturas probablemente sea el aspecto más descuidado. Se podría mejorar notablemente con una solución como la propuesta en esta tesis doctoral, la cual por su diseño genérico e independiente de la geometría texturizada sería perfectamente integrable en un sistema de ese tipo como en cualquier otro sin excesivos problemas.

2.8.2. Otros sistemas comerciales

A pesar de que la información sobre las técnicas utilizadas es muy limitada, resulta interesante realizar un pequeño análisis de productos comerciales como Google Earth[79], Microsoft Virtual Earth 3D[53], NASA World Wind[32], ESRI ArgGIS Explorer[7], Skyline TerraExplorer[26], Viewtec Terrainview[29] o Geovirtual GeoShow3D[76].

Google Earth

El producto de mayor difusión entre el público general es sin duda Google Earth. Este *software* está basado en el Earth Viewer, de la empresa Keyhole, comprada en 2004 por Google.

Maneja datos geoespaciales en un sistema de coordenadas geográficas basado en el datum WGS84 y dispone de datos a lo largo de todo el mundo con nivel de detalle muy diverso, aproximadamente entre 500m/texel y 0,15 m/texel según la zona.

Los requerimientos de *hardware* son bastante limitados, basta con una tarjeta gráfica con aceleración 3D y 16 MB VRAM, un PentiumIII 500 MHz, 128 MB de RAM y una conexión a Internet de 128 kbps. Este aspecto, junto con su difusión gratuita y su funcionamiento a través de Internet, han sido cruciales para el éxito que ha alcanzado.

Google Earth integra una enorme cantidad de información de diferentes fuentes con un elevado nivel de detalle. Además, se ha creado una herramienta muy potente con una interfaz de usuario agradable, fácil de usar y muy intutitiva. La navegación, la selección de capas de información, y la personalización de esta herramienta es muy cómoda y potente. Permite añadir localizaciones con datos adicionales como texto y fotografías, generar rutas entre lugares, incluir modelos 3D, etc. Además, estos datos pueden ser fácilmente compartidos entre los usuarios mediante el formato KML y publicados en Internet. De esta manera, se está creando alrededor de Google Earth una comunidad de usuarios y de compartición de información geoespacial sin precedentes.

Sin embargo, la calidad de las texturas y la presentación de la información SIG tiene serias limitaciones en calidad. El rendimiento también se podría mejorar notablemente, sobre todo en instalaciones donde se dispone de tar-

jetas gráficas modernas. La flexibilidad, escalabilidad para la visualización de información SIG, especialmente información dinámica también resulta mejorable. Estos aspectos podrían beneficiarse mediante la implantación de las técnicas desarrolladas en esta tesis doctoral.

Respecto al filtrado, Google Earth, a diferencia de otros productos, ofrece la posibilidad de realizar un filtrado anisotrópico de las texturas del terreno. Sin embargo, el resultado no es tan bueno como sería de esperar (figura 2.38). Otro problema que incrementa el *aliasing* es el hecho de utilizar líneas para representar la información vectorial, especialmente en vistas lejanas. Google Earth disimula bastante bien estos aspectos eliminando (con un fundido a transparente) los elementos vectoriales en la distancia e introduciendo niebla en las zonas lejanas.

Google Earth permite crear polilíneas, formadas por primitivas geométricas que se adaptan al terreno. Esto produce que en ocasiones algunas partes parezcan flotar sobre él y en otras ocasiones queden parcialmente enterradas. También provoca un notable *aliasing* cuando se observan a distancia.

De la misma forma, se pueden crear áreas sobre el terreno, pero en este caso no se adaptan a la superficie, sino que se forma un polígono con los vértices generados, del cual el usuario puede seleccionar la altura y las características de aspecto (color y nivel de transparencia del interior y del contorno, grosor del contorno, etc.), tal y como se muestra en la figura 2.39. Al desplazar la cámara, la gestión de LODs de geometría del terreno hace que la parte visible del polígono varíe.

También se permite incluir imágenes *raster* georreferenciadas sobre el terreno. En este caso, al tratarse de una textura sí se adaptan al terreno. La imagen se filtra correctamente al alejar la cámara, evitando la aparición de *aliasing*. Esta imagen se puede recibir de un servicio WMS o de un fichero local y actualizar periódicamente o en función de determinados parámetros como la visibilidad de la zona o que la cámara se detenga.

Microsoft Virtual Earth 3D

Microsoft Virtual Earth 3D visualiza la información SIG prerenderizada sobre los diferentes niveles de detalle de la textura. Tiene tres texturas disponibles: imagen aérea, mapa de carreteras e híbrida (que combina las dos anteriores). Esto incluye los textos, con el problema de que cuando el espectador no está orientado hacia el norte, su lectura se hace incómoda (ver figura 2.40). Además, el elevado nivel de compresión degrada mucho la calidad de las imágenes, lo cual se hace especialmente notable en los textos y los datos de origen vectorial como las líneas de fronteras políticas o carreteras.

El filtrado que se realiza sobre las texturas en vistas horizontales ofrece una calidad visual baja, las zonas lejanas a la cámara aparecen excesivamente borrosas. Este problema se aprecia en la figura 2.41 en las variantes de mapa de carreteras e imagen híbrida.

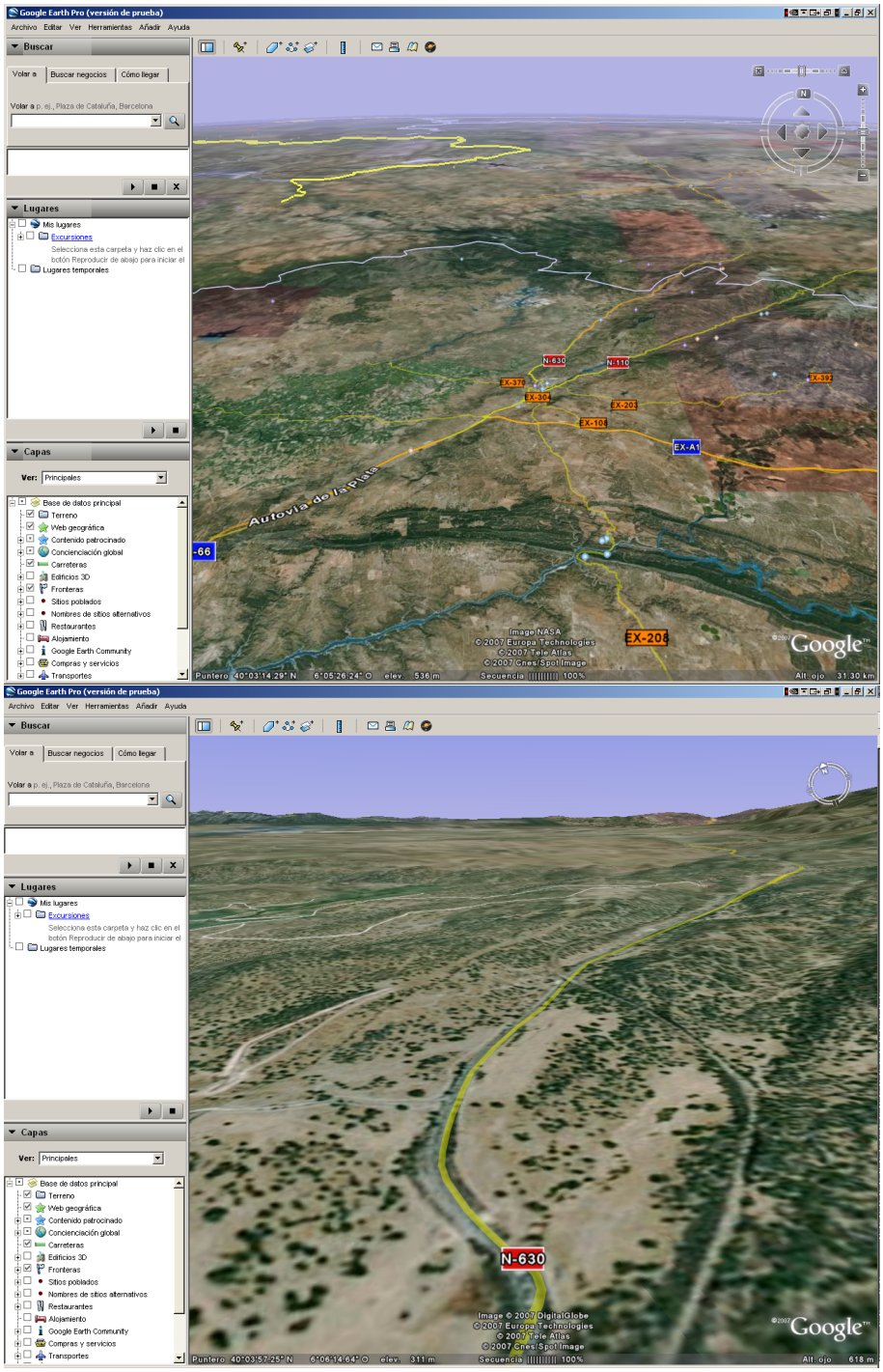


Figura 2.38: Filtrado anisotrópico (Google Earth Pro 4.2).

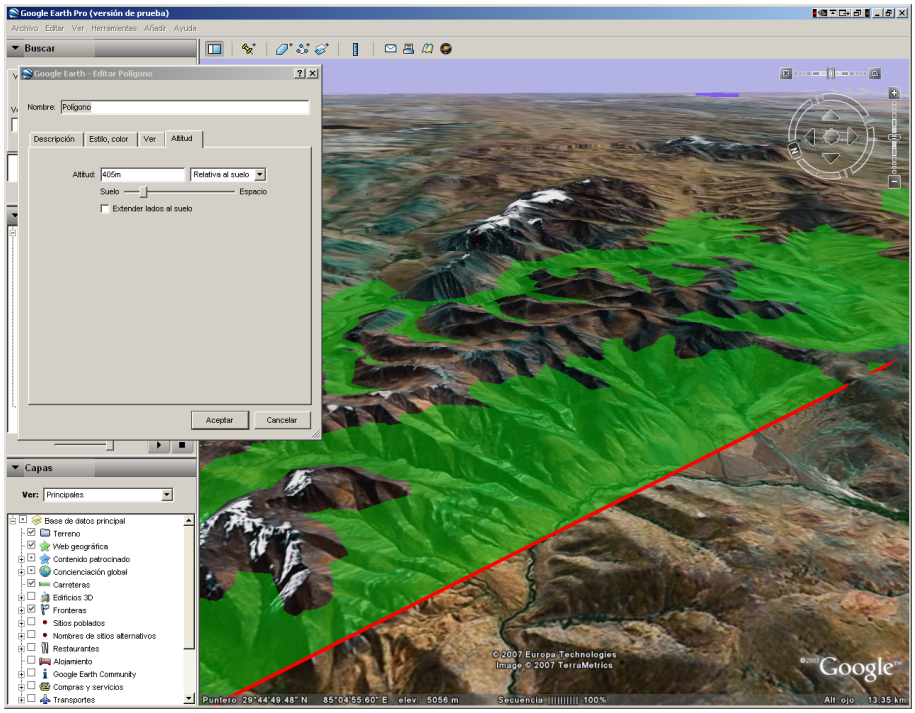


Figura 2.39: Creación de polígono plano sobre el terreno (Google Earth Pro 4.2).

Google Earth mejora el aspecto anterior, tratando las carreteras/fronteras, etc. como vectores. Por este motivo en ocasiones se aprecian errores de incoherencia entre la geometría del terreno y la información SIG, como que algunas líneas aparezcan flotando en el aire o se oculten enterradas bajo el suelo (ver figura 2.42).

NASA World Wind

NASA World Wind trata la información de manera similar a Virtual Earth 3D, mediante niveles de detalle de textura pregenerados que cambian de forma instantánea en un mosaico según la cámara se acerca o aleja del terreno. Sin embargo, esta textura se compone a partir de varias capas independientes que el usuario puede seleccionar a voluntad, de forma similar a Google Earth.

En World Wind se permite visualizar cierta información dinámica, como puede ser evolución de incendios o información meteorológica. El funcionamiento en estos casos consiste en la descarga de una serie de imágenes *raster* con los diferentes estados que se aplican secuencialmente junto con la imagen del terreno. Estas imágenes se solicitan a un servidor a través de WMS. La información está preprocesada, el volúmen de información y su resolución son muy limitados (ver figura 2.43).

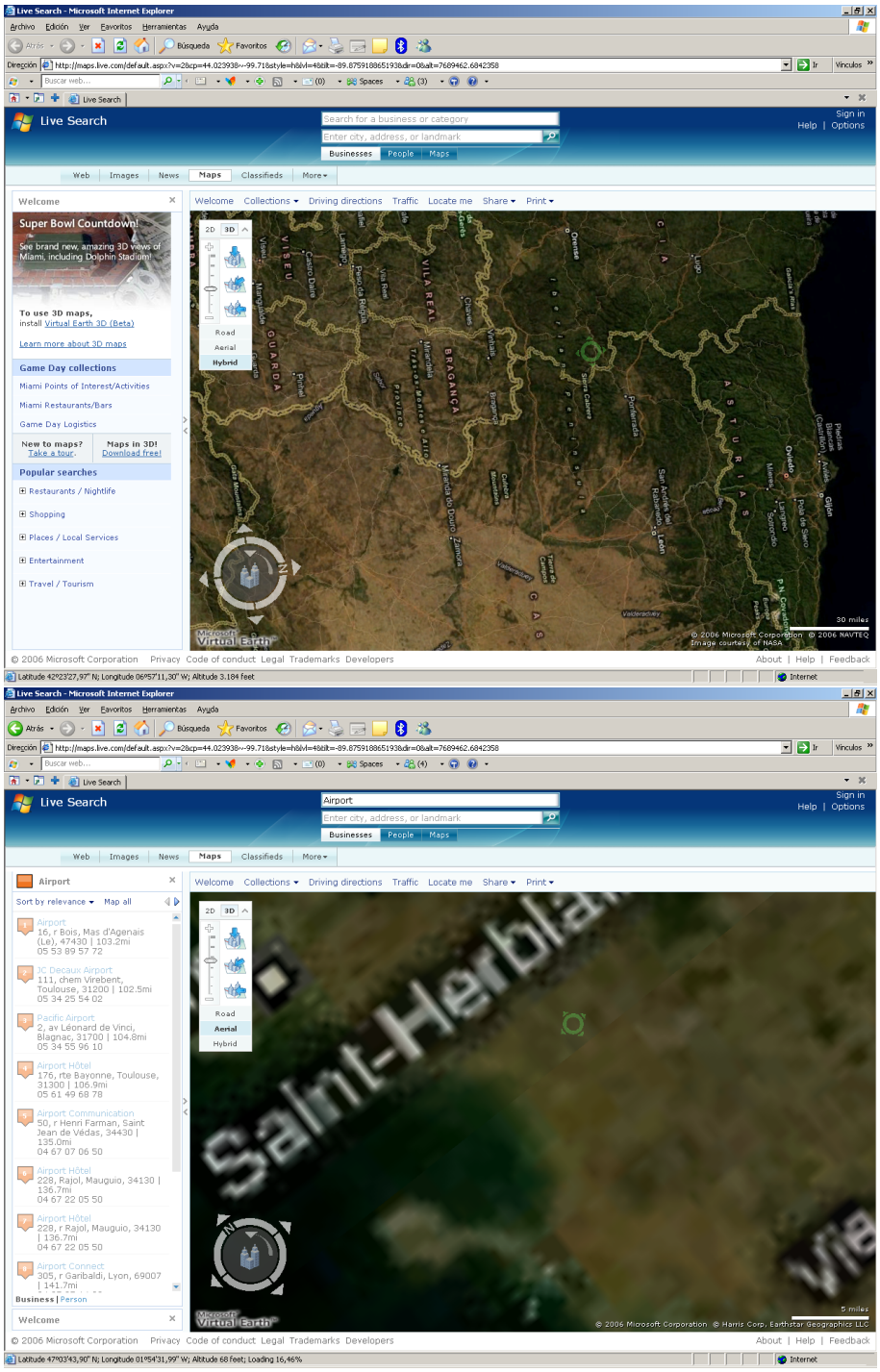


Figura 2.40: Visualización de la información SIG vectorial mediante textura (Microsoft Virtual Earth 3D).

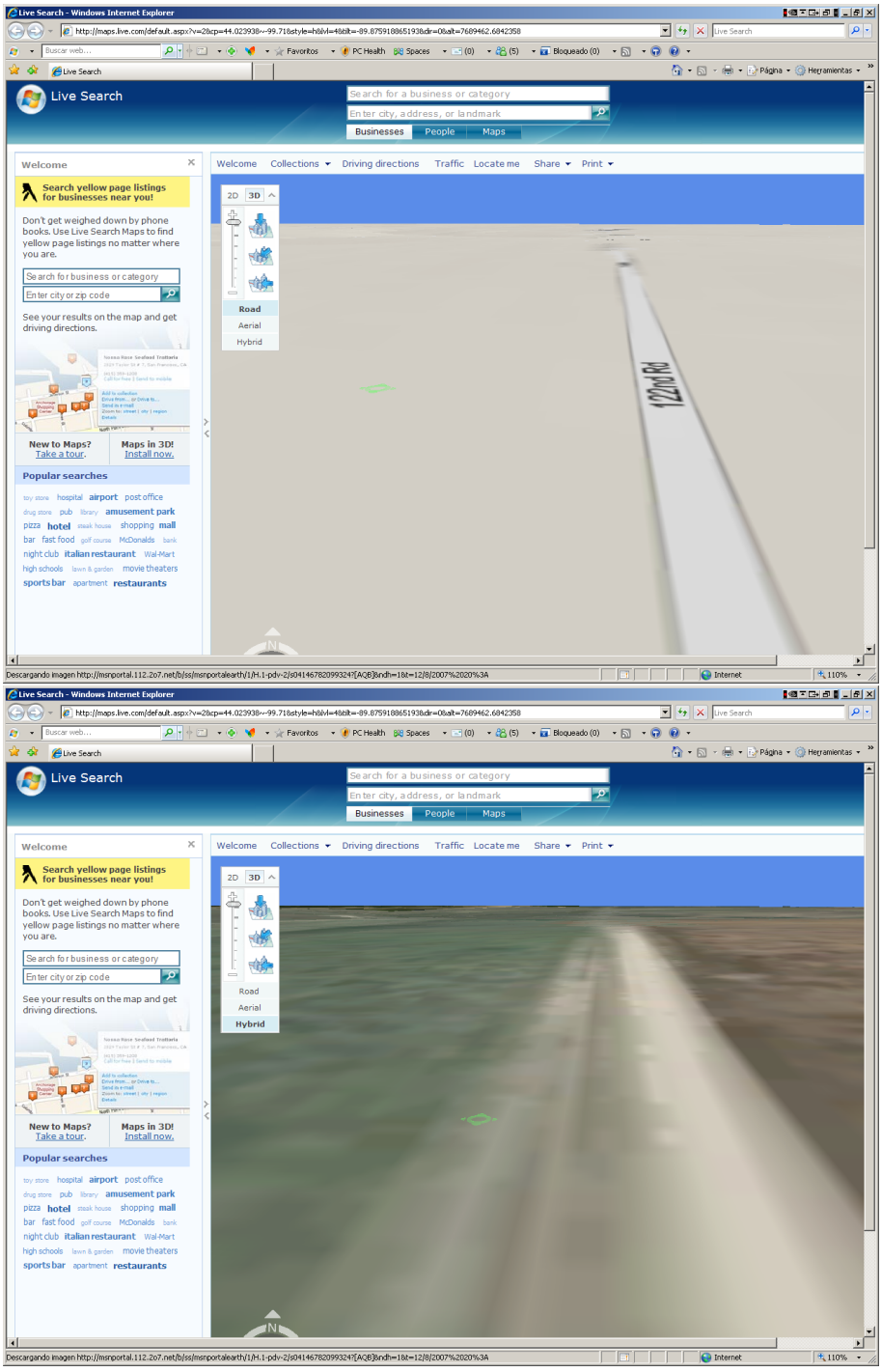


Figura 2.41: Problemas de *aliasing* en el texturizado (Microsoft Virtual Earth 3D).

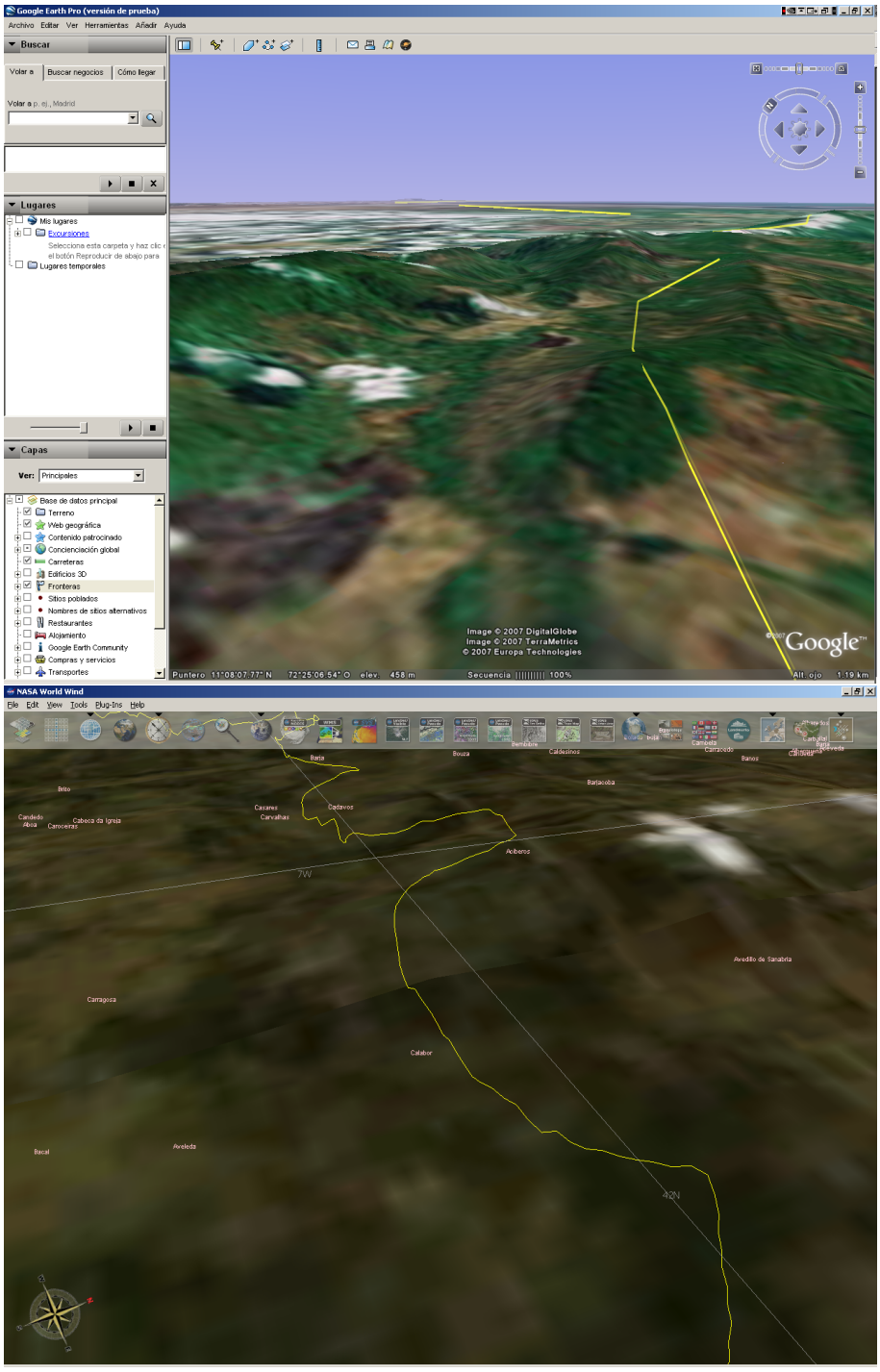


Figura 2.42: Problemas de visualización de la información SIG vectorial como geometría, (arriba) vectores atravesando el terreno, (abajo) vectores flotando sobre el terreno.

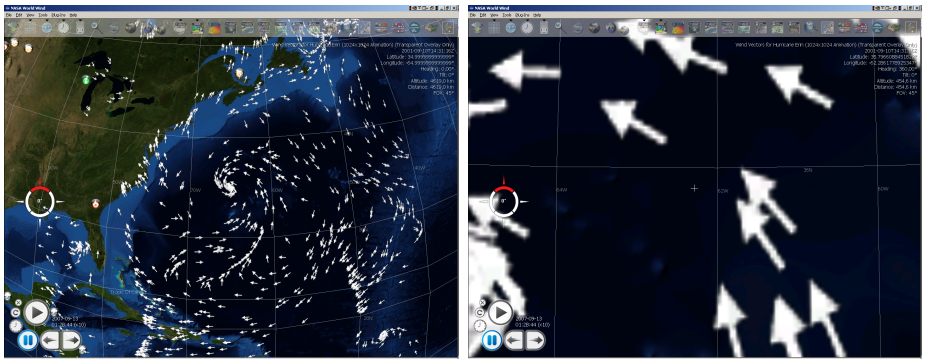


Figura 2.43: Visualización de información dinámica (NASA World Wind).

La NASA dispone de diversas bases de datos *on-line* con simulaciones y archivos históricos de datos georreferenciados que pueden visualizarse en World Wind. Estos datos están servidos por Internet a través de WMS. Además de información terrestre, disponen de información sobre otros astros, como Marte o la Luna.

Para la transmisión eficiente de las texturas utiliza el esquema de compresión DXT3. El *antialiasing* de texturas no es excesivamente bueno, parece que utilizan filtrado bilineal, lo cual provoca artefactos molestos en vistas lejanas, aunque de forma opuesta a la de Microsoft Virtual Earth 3D.

Aunque en alguna versión utilizó geometría para información como las fronteras políticas (ver figura 2.42, abajo), actualmente aplica esa información como una textura, con los consiguientes problemas de *aliasing* debidos a la resolución limitada. Sin embargo, parece que no combina esta textura correctamente con la imagen del terreno, sino que la aplica en una pasada posterior sin tener en cuenta la ordenación por profundidad. Esto produce efectos como el que se muestra en la figura 2.44.

El texto lo trata independientemente de la textura, al igual que Google Earth, pero en este caso parece que se gestiona con geometría 2D basada en líneas en lugar de utilizar una tipografía *raster*.

SkyLine Globe (TerraExplorer)

El TerraExplorer de Skyline tiene una calidad de filtrado reducida. Posiblemente realiza un filtrado bilineal dentro de cada resolución de la textura, pero no interpola entre niveles diferentes. El efecto es mejor en las zonas lejanas. Aunque la carga de los niveles es repentina, y este efecto desagradable empeora aún más con la carga desordenada de los trozos del mosaico de textura, una vez cargados los niveles realiza transiciones suaves entre ellos, lo cual es una mejora en la calidad de la visualización respecto a la mayoría de los otros sistemas.

La información vectorial basada en polilíneas, como las carreteras, se

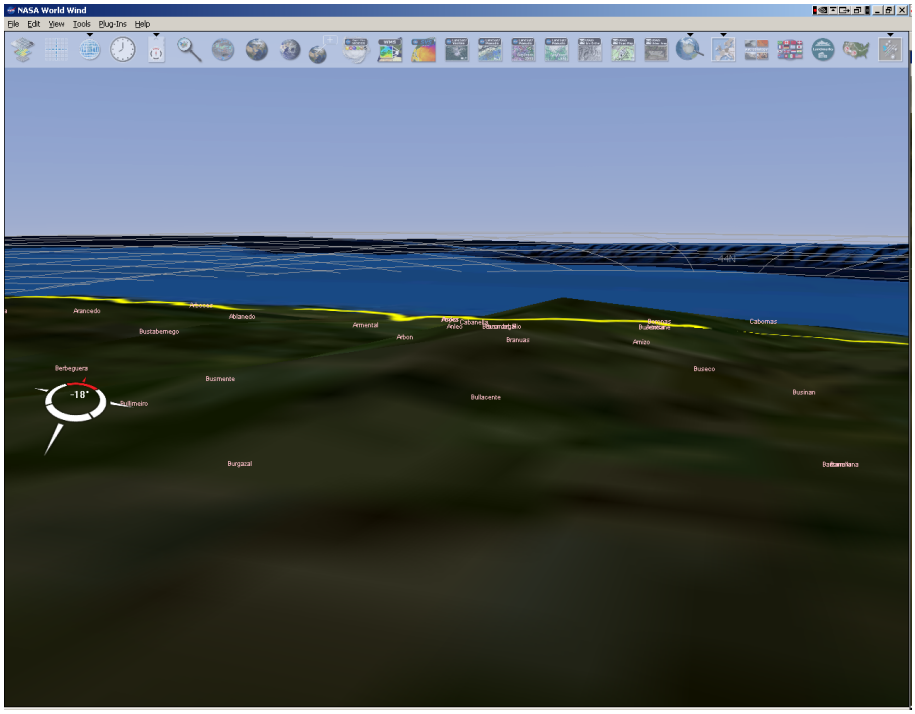


Figura 2.44: Visualización de información vectorial *rasterizada* sin ordenar en profundidad (NASA World Wind).

gestiona mediante texturas. Esto se puede apreciar al cargar los niveles, porque aparece la imagen borrosa en los niveles más burdos antes de que se reciban los de mayor detalle.

Uno de los aspectos más descuidados es la baja calidad de visualización en el mar, donde se aprecia un efecto de patrón de textura repetida a diferentes niveles de detalle. Esto se podría mejorar enormemente sólo con aplicar un color sólido a la zona del mar.

Permite marcar un área sobre el terreno, genera dinámicamente la polilínea que rodea ese área en una textura multirresolución y la rellena de un color transparente. También permite dibujar a mano alzada sobre el terreno, de forma similar a las polilíneas, generando la textura multirresolución dinámicamente y mapeándola sobre el terreno. El rendimiento es bastante bueno.

La versión de pago permite la creación y edición de geometría 2D y 3D dentro del terreno así como texto, imágenes y vídeo. También permite la inclusión de lo que denomina objetos dinámicos, que consiste en elementos puntuales representados mediante texto, imágenes o modelos 3D que siguen una ruta definida por el usuario. También se puede utilizar esta ruta para seguirla con la cámara en primera persona.

Es posible también la conexión con un GPS para recibir los datos de posición en tiempo real o a partir de datos grabados. La ruta correspondiente se muestra mediante el movimiento de un icono determinado sobre el terreno.

ViewTec TerrainView

ViewTec TerrainView es un *software* similar a los analizados, que permite visualizar terreno realista en 3D navegando de forma interactiva. La fluidez del movimiento es notablemente buena. Uno de los aspectos más destacables de TerrainView es que incluye un motor de simulación de nubes con un aspecto bastante realista, lo cual para algunas aplicaciones puede ser de gran utilidad.

El filtrado no es excesivamente bueno, pero la transición entre niveles de detalle de textura es muy suave. Los textos se muestran de forma independiente a las texturas, mediante geometría, con aspecto similar a World Wind.

TerrainView está preparado para la inclusión de modelos 3D de poblaciones y vegetación, permitiendo bajar la cámara desde la vista de pájaro hasta el nivel del suelo.

Sin embargo, la capacidad de representación de información vectorial 2D sobre el terreno 3D no está muy trabajada. Los vídeos de demostración no van más allá de la ubicación de textos con los nombres de las poblaciones a la vista de la cámara y en algún caso contornos y fronteras políticas vistas a cierta distancia. Existe la posibilidad de importar algunos formatos vectoriales, como los ESRI Shapefiles (.shp), pero en cualquier caso no maneja

Figura 2.45: Visualización de información vectorial mediante geometría (Geovirtual GeoShow3D).

información dinámica.

Geovirtual GeoShow3D

La empresa española GeoVirtual desarrolla el *software* GeoShow3D, un navegador de terreno 3D interactivo que permite la importación de capas SIG, tanto *raster* como vectoriales, estas últimas gestionadas mediante geometría, con los consiguientes problemas (figura 2.45).

El aspecto más negativo de este sofware es la navegación que resulta incómoda y poco intuitiva.

El filtrado se puede configurar, activando uso de *mipmapping* y definiendo el nivel anisotropía, para reducir el efectos de *aliasing* en el terreno. Aún así, el rendimiento y la calidad son bastante pobres en comparación a otros sistemas similares.

Dispone de herramientas de medición que permiten marcar interactivamente polilíneas o áreas, que se visualizan mediante geometría, pero esta información desaparece al poner en movimiento la cámara.

El modo de funcionamiento de este *software* se basa en la creación de los mundos mediante una herramienta de edición de la misma compañía que produce como resultado una fichero visualizable únicamente con GeoShow3D. Por lo tanto, no se trabaja con información dinámica ni se permite el acceso

a servidores utilizando protocolos estándar como los definidos por el OGC.

ESRI ArcGIS Explorer

ArcGIS Explorer es la aplicación ligera para la visualización de información geográfica sobre terreno 3D integrada con el paquete de *software* ArcGIS de ESRI. En aspecto y funcionamiento es bastante similar a Google Earth. Está preparado para tomar los datos del servidor global de ESRI a través de Internet o de cualquier otro servidor de ESRI accesible por el usuario.

El filtrado de la textura es inadecuado (ver figura 2.46). Por el aspecto se debe tratar de un filtro bilineal, lo cual produce un *aliasing* bastante grave en vistas no cenitales. En este aspecto, ArcGIS Explorer es notablemente peor que los otros sistemas similares analizados.

La información vectorial se gestiona mediante textura. Se aplica de forma muy similar a como lo hace Microsoft Virtual Earth 3D. La información está pregenerada en textura con contenidos diferentes en cada nivel de detalle. Los niveles se activan por altura y no por distancia del terreno a la cámara, por lo que en alturas bajas las zonas lejanas tienen un nivel de detalle elevado, provocando un fuerte y molesto *aliasing*. Además, tiene el mismo problema con los textos cuando la vista no está orientada hacia el norte.

Permite crear anotaciones sobre el terreno en forma de líneas y polígonos, que convierte en textura con diferentes resoluciones y aplica al terreno sincronizada con la textura base en cuanto a nivel de detalle. Los polígonos se muestran mediante geometría durante la creación interactiva (figura 2.47). Una vez definido, se convierte en una textura que se aplica mapeada al terreno. La generación de esta textura es excesivamente lenta, la espera puede llegar a ser de varios segundos, lo cual, teniendo en cuenta que es un contenido generado localmente, no descargado por la red, no es justificable.

Sin embargo, en el caso de crear polilíneas, se manejan como geometría y no como textura, no sin notables problemas de precisión y sincronización con la superficie del terreno. En la figura 2.47 (inferior) se aprecian estos problemas.

La mayor ventaja de este *software* es su buena integración con un completo conjunto de herramientas SIG, como es ArcGIS de ESRI, especialmente servidores de datos geoespaciales. Además, no se limita a los servidores propios, sino que utiliza protocolos estándar para acceder a otros servidores de datos geoespaciales, tanto de otras compañías como *software* libre.

Sin embargo, las características de calidad y rendimiento en cuanto a la visualización de los datos vectoriales sobre el terreno es altamente mejorable. El producto podría verse enriquecido implementando la técnica descrita en esta tesis doctoral.

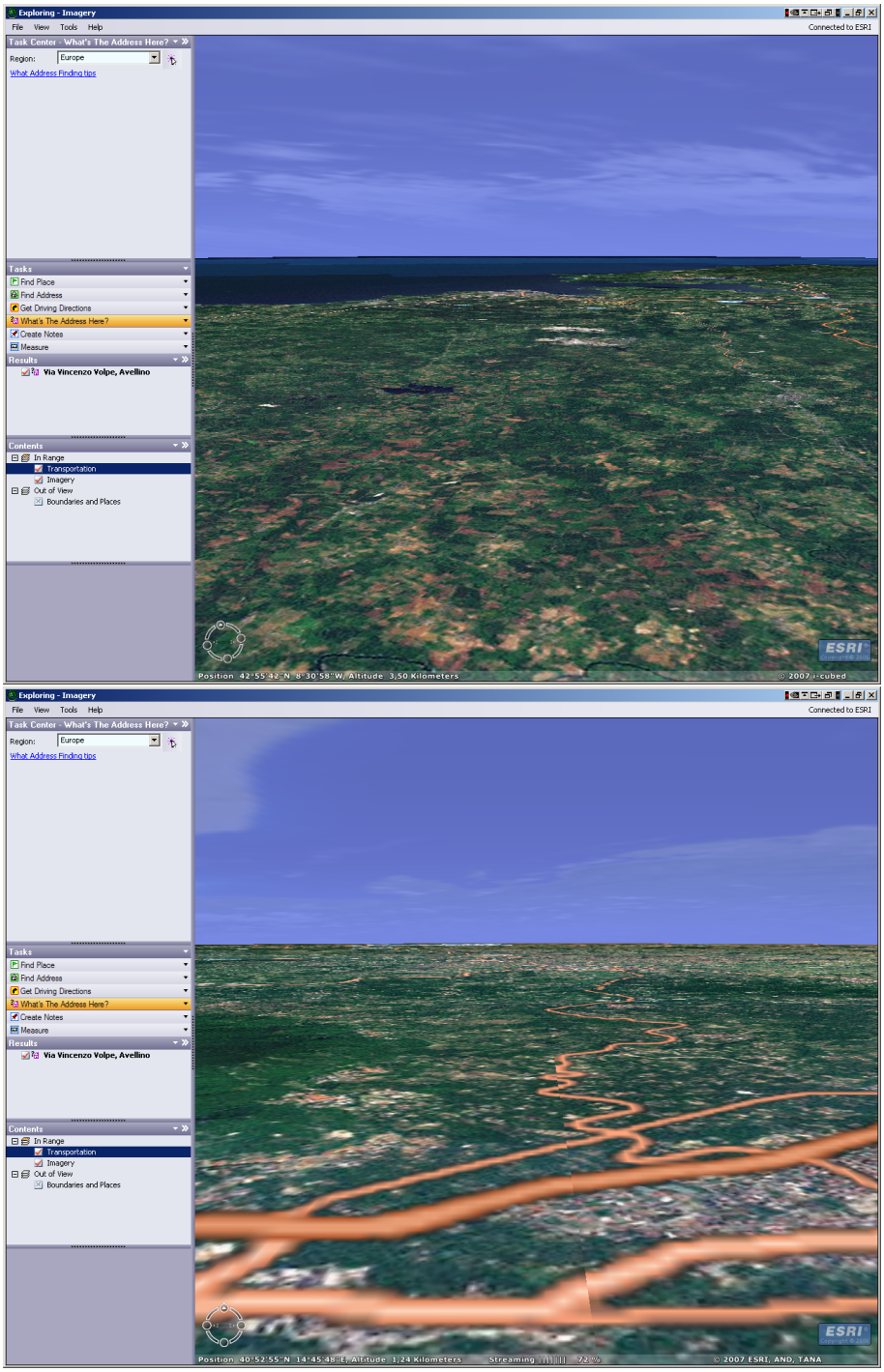


Figura 2.46: Problemas de *aliasing* por el filtrado bilineal de las texturas (ArcGIS Explorer).

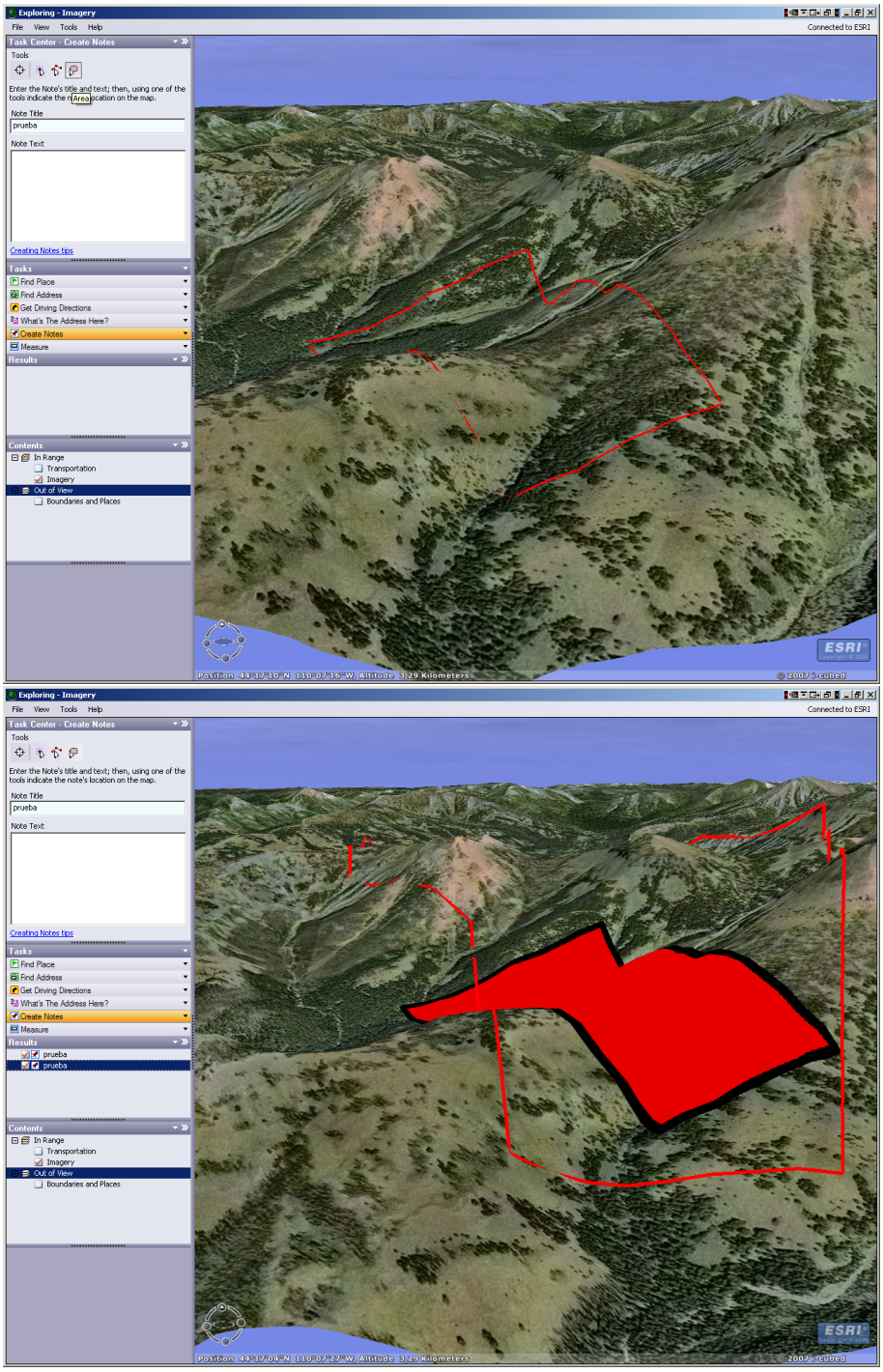


Figura 2.47: Creación interactiva de polígonos sobre el terreno (ArcGIS Explorer). Arriba: edición interactiva mediante geometría. Abajo: Resultado final mediante textura.

Conclusiones

Los mayores problemas detectados en el funcionamiento de las aplicaciones estudiadas son los siguientes:

- En general, las transiciones entre los diferentes niveles de detalle son muy bruscas, llegando a aparecer o desaparecer de golpe ciertos elementos. No se realizan transiciones suaves entre niveles. En muchos casos, ni se realizan fundidos entre los niveles, ni filtrado trilineal ni mucho menos anisotrópico.

- La partición en mosaico de imágenes se hace muy evidente debido a la carga y presentación repentina de estos elementos sueltos en un orden aparentemente aleatorio (figura 2.48). Algunos sistemas, como World Wind, hacen patente esta división incluso marcando las casillas del mosaico durante la carga (ver figura 2.49).

- El trabajo con fragmentos separados de textura produce uniones visibles (rotura de la coherencia espacial[55]). Este problema se acentúa debido a la fuerte compresión de estas texturas independientes (ver figuras 2.50 y 2.51).

- Aunque existen versiones más avanzadas del *software* que pueden trabajar con información SIG, habitualmente la única funcionalidad que se ofrece es un módulo de importación de datos SIG, por lo que no permite trabajar directamente con información dinámica mediante acceso a servidores de datos geoespaciales. World Wind es uno de los pocos que permiten el acceso mediante WMS a información remota dinámica, pero se limita a la descarga de una secuencia de imágenes *raster* de resolución limitada que aplica como texturas. Otra capacidad existente en algunos de los productos analizados es la importación de datos de GPS (principalmente rutas y localizaciones) y en algunos casos incluso la conexión en tiempo real para obtener la posición del GPS. Pero esta información se limita a la ubicación de elementos puntuales o el seguimiento de la posición de un punto.

- En otros casos, se permite crear primitivas sencillas, como ya se ha descrito, pero las posibilidades son muy limitadas y el rendimiento y la calidad bastante pobres.

- La cantidad de información visualizada sobre el terreno tiene serias limitaciones. Por ejemplo, la versión Google Earth Pro 4.2 está limitada a 2.500 ubicaciones o coordenadas geoespaciales.

- El hecho de tomar las imágenes de fuentes muy diversas provoca que la vista del terreno tenga un aspecto poco realista, de mosaico. Se produce una continua rotura de la coherencia espacial debido a las diferencias

de aspecto de las imágenes, principalmente en color y resolución. Este efecto se plasma en la figura 2.52.

- En muchos casos, la calidad del filtrado de las texturas es bastante pobre. En algunos casos se aprecia más en los acercamientos (imagen borrosa) y en otros casos, como Google Earth, al alejar la cámara (vibraciones y parpadeos por información de alta frecuencia).

- En algunos casos, la geometría del terreno produce cambios bruscos e incluso se llegan a producir errores geométricos en las uniones de fragmentos de terreno (ver figura 2.53).

2.8.3. Resumen de las técnicas estudiadas

La aplicación de los datos *raster* a grandes extensiones de terreno con alta resolución tiene una problemática compleja que ha sido afrontada por numerosos trabajos en las últimas décadas. Para conseguir una respuesta interactiva adecuada se utiliza la capacidad de mapeado de textura del *hardware* gráfico, disponible en la totalidad de los sistemas actuales. Sin embargo, estas texturas tienen una severa limitación en tamaño. Incluso las GPUs actuales que han elevado el límite hasta 8192^2 texels, resultan insuficientes para este tipo de aplicaciones.

Cosman [55] planteó las bases de lo que denominó "textura de terreno global", definiendo las dos estrategias posibles: mosaico de texturas y *hardware* específico para la gestión de un espacio virtual de textura de tamaño ilimitado.

Tanner[144], siguiendo la filosofía de Cosman, definió un sistema muy eficiente y adecuado para resolver el problema del texturizado de terreno, con las mismas virtudes que el de Cosman, principalmente la independiencia de la geometría, pero también con la misma grave limitación: la necesidad de un *hardware* específico de elevado coste.

Posteriormente se publicaron otras técnicas con requerimientos menos exigentes. Una de las primeras fue la de Rabinovich [123], que cacheaba un subconjunto de una imagen más grande en una textura del sistema gráfico. Los problemas de esta técnica eran que estaba limitada al espacio de textura disponible en el *hardware* y que la actualización de esa textura resultaba lenta y costosa, por tener que desplazar una gran cantidad de memoria para seguir el movimiento de la cámara. Este problema sería resuelto por Tanner posteriormente mediante el direccionamiento toroidal tal y como se ha descrito en la sección 2.6.3.

La mayoría de las técnicas posteriores se basan en un mosaico de texturas, con todos los problemas subyacentes, anteriormente descritos por numerosos autores, especialmente Cosman en su trabajo pionero en el tema [55]. El mayor de estos problemas, sin duda, es el fuerte acoplamiento entre la

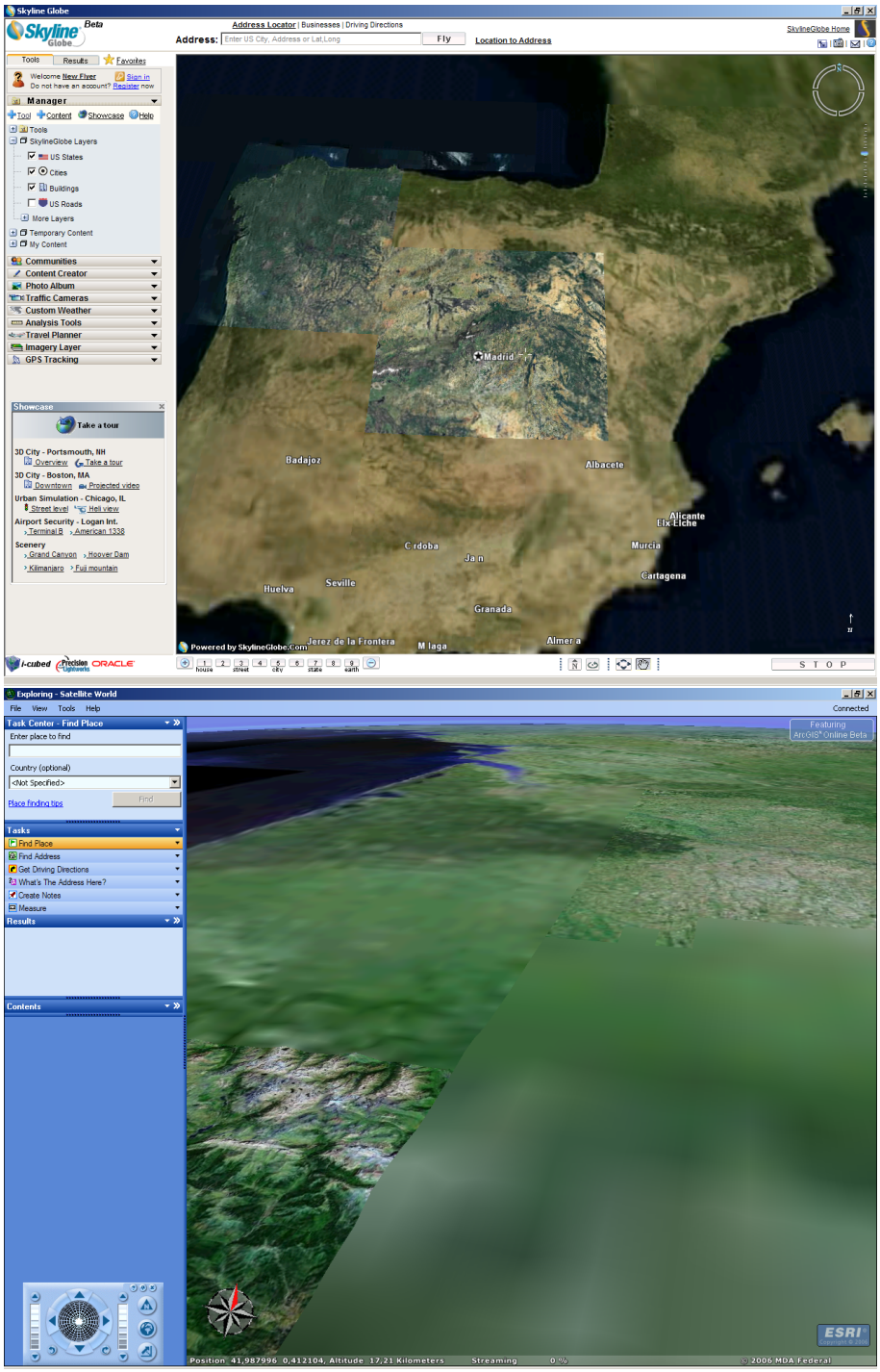


Figura 2.48: Carga desordenada de niveles del mosaico de texturas.

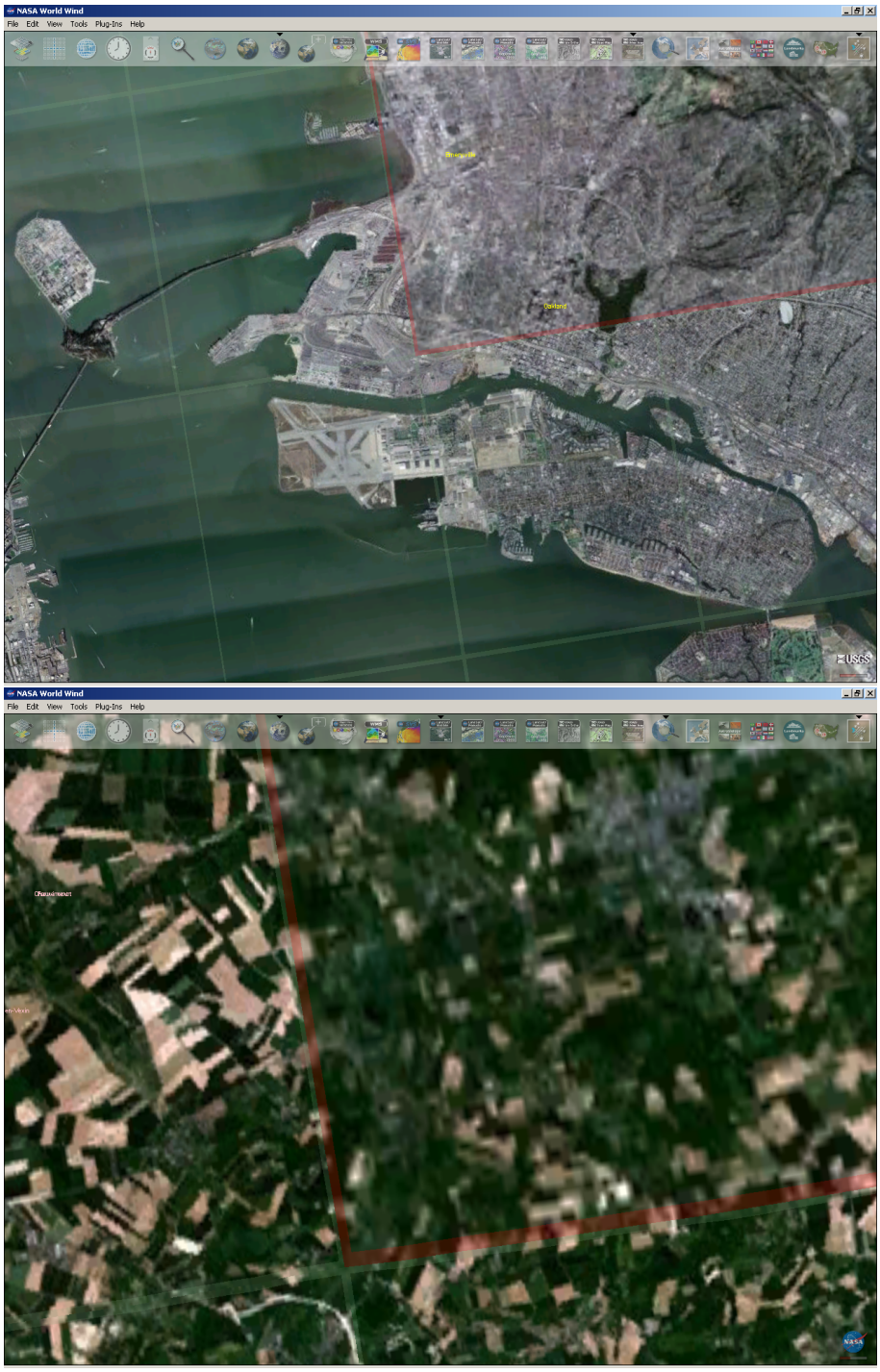


Figura 2.49: Carga desordenada de niveles del mosaico de texturas (NASA World Wind).

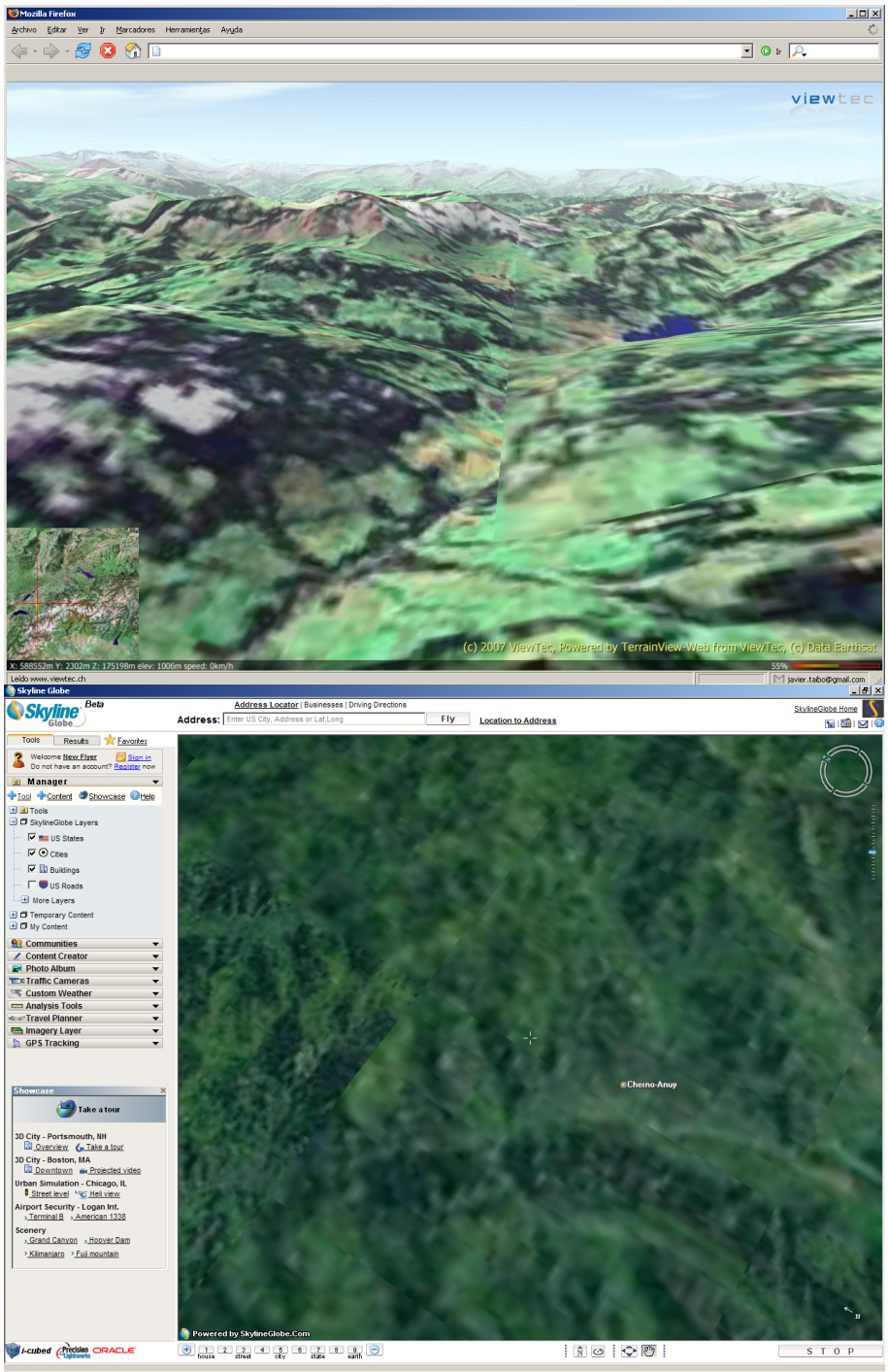


Figura 2.50: Rotura de la coherencia espacial en las uniones de texturas.

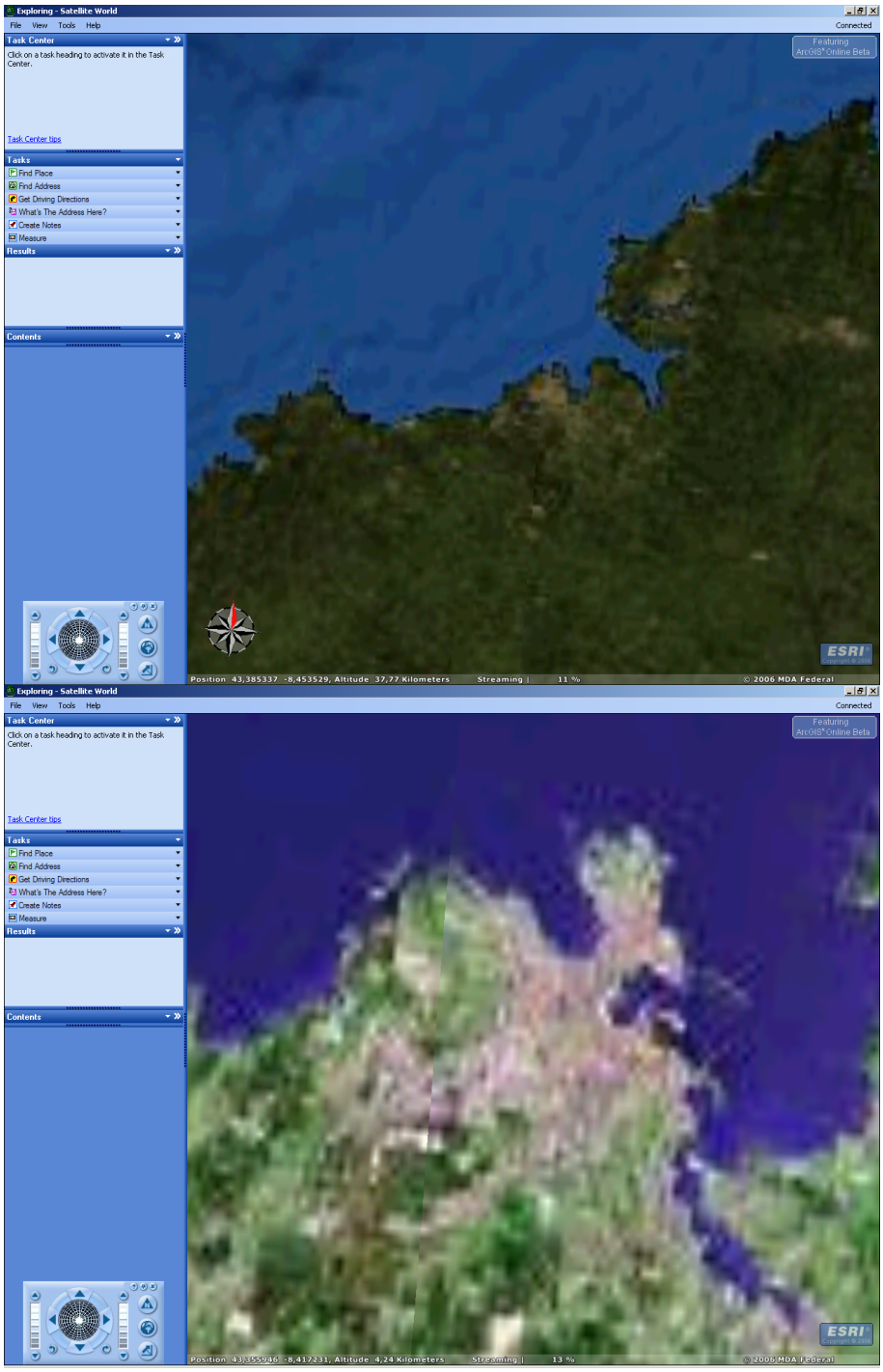


Figura 2.51: Rotura de la coherencia espacial en las uniones de texturas (continuación).

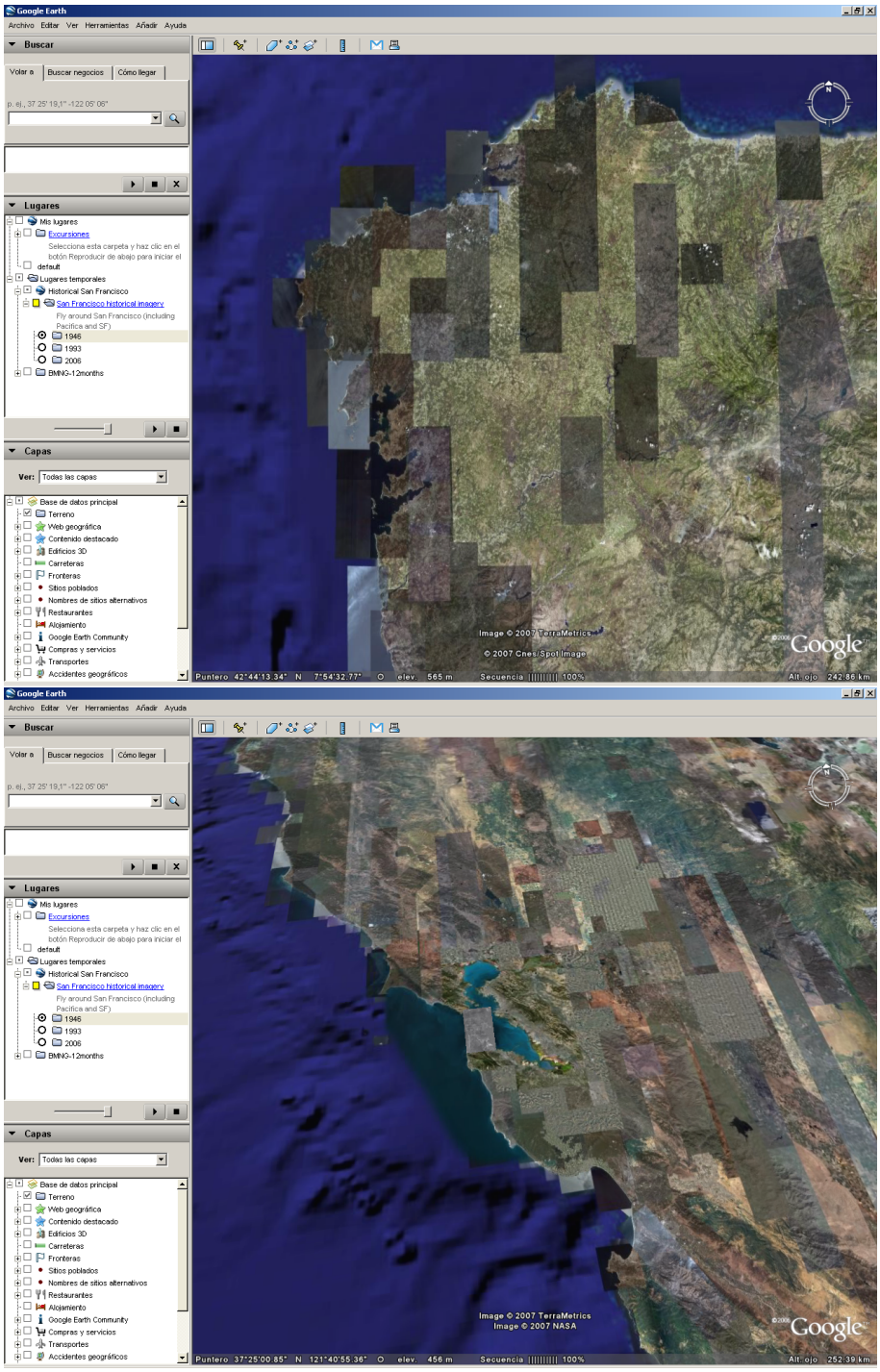


Figura 2.52: Rotura de la coherencia espacial debido al montaje de imágenes de distintas fuentes sin ecualizar.

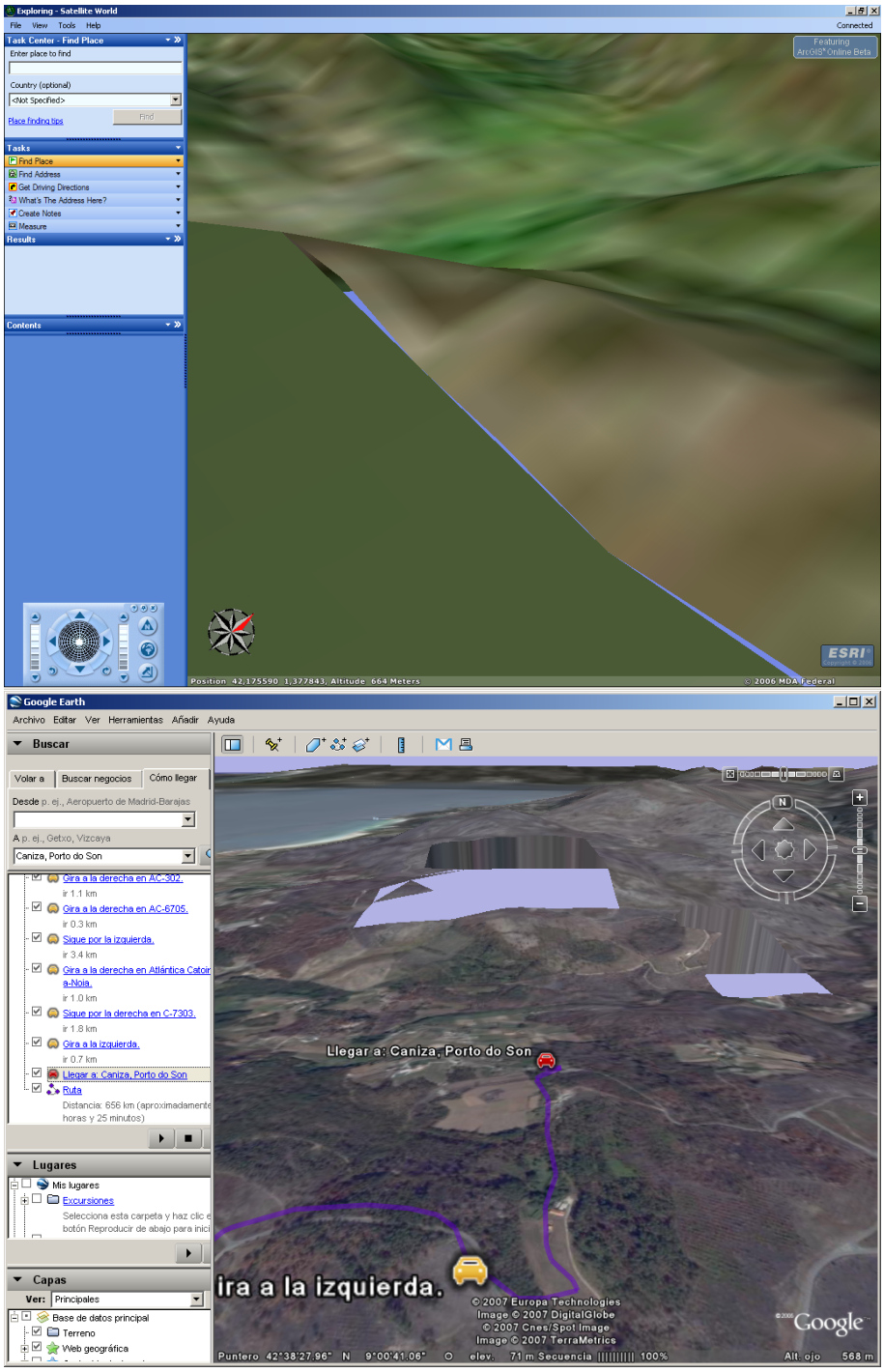


Figura 2.53: Problemas en la geometría del terreno.

geometría y la textura en dos aspectos principalmente: organización y estructuración de los datos y gestión de los LODs. Esto limita enormemente la libertad del motor de geometría y el nivel de detalle que se puede alcanzar tanto en geometría como en textura. Se dificulta especialmente la organización de los datos de elevación del terreno en estructuras irregulares (TINs) y la simplificación de polígonos más allá del tamaño de bloque de textura.

Estos problemas se encuentran presentes en las técnicas de Hüttner[92], Cline [52], Döllner [65], Klein[98], Cignoni[49, 50], Losasso[107], Holkner[89] y Brodersen[44], descritas en este capítulo.

En trabajos posteriores, como los de Ephanov[68] y Seoane[137], se reduce este acoplamiento siguiendo una filosofía similar a la de los *clipmaps*, adaptándolos a las limitaciones del *hardware* gráfico disponible. Sin embargo, no se llega a eliminar por completo, y el uso de una fuerte simplificación de la geometría provoca una caída en el nivel de la textura.

Respecto al filtrado de la textura para evitar los problemas de *aliasing*, muchas de estas técnicas utilizan la capacidad de filtrado trilineal del *hardware* gráfico, basado en *mipmaps*. En algunos casos, por el propio diseño del sistema, se impide el uso de este filtrado trilineal por *hardware*, como es el caso de los Geometry Clipmaps de Losasso y Hoppe [107], que sincroniza los LODs de textura y geometría realizando un fundido en las uniones, o la técnica de Holkner [89] que también está limitado a filtrado bilineal.

La mayoría de las técnicas, excluyendo Cosman y Tanner, no tienen requisitos especiales de *hardware* más allá de las características soportadas en OpenGL 1.2. Las excepciones son la de Losasso[107] y la de Holkner[89] que precisan de GPUs con una cierta programabilidad. Döllner [65] sugiere el uso de la capacidad multitextura en caso de estar disponible y las técnicas de Ephanov [68] y Seoane [137] plantean el uso de *shaders* en caso de estar disponibles para mejorar ciertos aspectos de estas técnicas, aunque no resultan imprescindibles. Además, estas dos últimas técnicas facilitan por su diseño el uso de las texturas virtuales dentro de otros shaders para combinarlas en efectos más complejos.

En la tabla 2.6 se resumen los principales aspectos de las técnicas estudiadas, incluyendo el acoplamiento con la geometría, tipo de filtrado, capacidad de combinar varias texturas, gestión de bases de datos extensas y necesidades de *hardware*.

En cuanto a la representación de los datos vectoriales 2D sobre el terreno 3D, se han estudiado técnicas que se enfocan en la construcción de geometría 3D, como las de Wartell [146], Schneider [133], Agrawal [33] y Schilling [132], con los problemas asociados a este tipo de estrategia, mencionados anteriormente. Principalmente limitaciones debido a la complejidad de los cálculos, especialmente en combinación con LODs geométricos de terreno y/o con información vectorial dinámica, y problemas en el *render* de polígonos coplanares.

La otra alternativa habitual es la proyección de los datos vectoriales en

Técnica	*Año*	*Acoplamiento*	*Filtrado*	*Multitextura*	*Shaders*	*Out-of-core*	*HW específico*
Cosman[55]	1994	NO	Tri	NO	NO	SI	E&S
Rabinovich[123]	1997	NO	Tri			SI	NO
Clipmaps[144]	1998	NO	Tri	NO	NO	SI	SGI
MP-Grid[92]	1998	SI	Tri				NO
Cline[52]	1998	SI	Tri				NO
Döllner[65]	2000	SI		SI			NO
Klein[98]	2002	SI					NO
(P-)BDAM[49, 50]	2003	SI					NO
G. Clipmaps[107]	2004	SI	acopl				shader
Holkner[89]	2004	SI	Bi				shader
Brodersen[44]	2005	SI	HW				NO
Virtual Textures[68]	2006	poco	Tri/ani	SI	SI		NO
Seoane[137]	2007	poco	Tri/ani	SI	SI	SI	NO

Cuadro 2.6: Comparativa de técnicas de texturizado de terreno.

una textura que se aplica al terreno. Esta estrategia ha sido utilizada en algunas técnicas como las de Kersting [96], Schneider [133] y Brooks [45].

La gran ventaja del uso de texturas es que desacopla completamente la información vectorial de la geometría del terreno y por lo tanto no se ve afectada por la complejidad o las variaciones de dicha geometría. Además, se eliminan los problemas de precisión en *Z-buffer* en el *render* de polígonos coplanares.

La principal limitación que los defensores de las técnicas basadas en geometría achacan al uso de texturas es que la información vectorial ofrece poca calidad debido a la resolución limitada de las texturas.

La técnica de Kersting propone la generación bajo demanda de las texturas al LOD adecuado para las condiciones de visualización. Sin embargo, no se trata de forma adecuada el filtrado de la textura para evitar el *aliasing*. El hecho de trabajar con un único LOD limita las posibilidades a técnicas de filtrado por proximidad o bilineal.

Brooks describe un SIG que combina vistas en 2D y 3D de la información vectorial aplicada como textura. Sin embargo, se limita a información estática y a un área limitada, o al menos no se describe en el artículo cómo resuelven estos problemas.

La técnica de Schneider [133] tiene la peculiaridad de que combina geometría y textura, aprovechando las ventajas de cada estrategia. Trocea el terreno y lo organiza en una estructura de *quadtree*, tanto para la geometría

Técnica	Año	Acoplamiento	Aproximación	Info dinámica	Terreno dinámico	Primitivas
Kersting[96]	2002		T	SI	SI	
Wartell[146]	2003	SI	G	NO		PL
Brooks[45]	2005		T	NO	NO	
Schneider[133]	2005	SI	G/T	SI(tex)	SI	
Agrawal[33]	2006	SI	G	NO		PL
Schilling[132]	2007	SI	G	NO		
Schilling[132]	2007	NO	S	SI	SI	

Cuadro 2.7: Comparativa de técnicas de visualización de información SIG 2D sobre el terreno 3D.

como para la textura que van de esta forma completamente acopladas. Las texturas se generan bajo demanda y la geometría se pregenera en un proceso off-line, por lo que se limita a información estática.

La última técnica estudiada, también de Schneider [134], utiliza una aproximación original mediante el *stencil buffer*, pero dispara el rendimiento cuando el número de capas es elevado y/o la complejidad geométrica es alta. Además, sufre limitaciones en cuanto al aspecto que se puede dar a los elementos vectoriales (básicamente las opciones se reducen a un color sólido).

En la tabla 2.7 se resumen las características de las técnicas de visualización interactiva de información SIG vectorial sobre modelos 3D de terreno, indicando si existe acoplamiento con la gestión de geometría, qué tipo de aproximación se utiliza (geometría, textura, *stencil buffer*), si se maneja información vectorial dinámica, si la geometría del terreno es dinámica y si está limitado a algún tipo de primitiva vectorial.

Tras este estudio y análisis de los trabajos previos, se puede concluir que no existe actualmente ninguna técnica que solvente todos los problemas planteados. Por este motivo, se aborda en esta tesis el desarrollo de una nueva técnica que resuelva dichos problemas y cumpla los objetivos planteados. Estos objetivos se describen en el siguiente capítulo.

Capítulo 3

Hipótesis

3.1. Punto de partida

Tras el estudio del estado del arte y un análisis detallado de los trabajos e investigaciones previas en el tema de la visualización combinada de información SIG 2D con modelos digitales 3D de terreno, se propone un enfoque novedoso para resolver el problema. Se soluciona de una manera flexible y que ofrece una buena calidad sin sacrificar en ningún momento el rendimiento necesario para una aplicación interactiva exigente, como puede ser un simulador de vuelo.

Uno de los principales aspectos detectados en el estudio de las técnicas existentes ha sido un fuerte acoplamiento de la gestión de texturas con el tratamiento del modelo geométrico del terreno. Este problema se produce en todas las técnicas implementables en hardware de bajo coste, y su solución se ha planteado como uno de los principales objetivos del presente trabajo.

La información visualizada puede ser dinámica, de diferente naturaleza (*raster* y vectorial) y procedente de múltiples fuentes, tanto locales como remotas. Puede también estar en diferentes formatos, sistemas de coordenadas, tipos de proyección, *data* geográficos, etc. Toda esta amalgama de información debe ser perfectamente ubicada con la precisión adecuada sobre la representación 3D del terreno.

Las técnicas desarrolladas en esta tesis doctoral forman parte de un proyecto completo de visualización de terreno [83]. Una de las decisiones de diseño que se tomaron respecto a la arquitectura general del sistema es la separación entre las tareas de gestión de los diferentes tipos de datos geoespaciales visualizados. Estos datos incluyen los siguientes:

- **Modelo geométrico de elevación del terreno** (al cual nos referimos habitualmente como "geometría" del terreno). Puede estar basado en diferentes técnicas, incluyendo el uso de mallas regulares (*grids*) o redes de triángulos irregulares (*TINs*), organizaciones jerárquicas como *quadtrees* [130], *quadtrees* esféricos [72], *bintrees* [66], *btrees* [31], etc.

- **Imágenes *raster*** provenientes de diferentes fuentes, principalmente fotografía aérea e imagen de satélite, aunque también podría consistir en cualquier mapa temático (usos de suelo, mapas geológicos, estudios de cobertura radioeléctrica, mapas demográficos, etc).

- **Datos vectoriales bidimensionales**, generalmente provenientes de SIG o CAD. Pueden consistir en información almacenada en bases de datos, ser resultado de cálculos realizados sobre otros datos o incluso recibirse directamente desde dispositivos de medición en tiempo real como estaciones meteorológicas, dispositivos de posicionamiento global (*GPS*) instalados en vehículos, medidores de niveles de tráfico, etc. Además, estos datos espaciales pueden tener asociada información no espacial (atributos) que deberán igualmente ser representados.

- **Modelos tridimensionales** de elementos de interés: infraestructuras, edificaciones, obras de ingeniería civil, vegetación, o cualquier otra información sobresaliente del terreno.

- **Información volumétrica**. Además de los modelos construidos a partir de superficies o sólidos, resulta de gran interés el manejo de información de tipo volumétrico, considerando la división del espacio tridimensional en celdas denominadas *voxels*, sin limitarse a la superficie del terreno. Ejemplos de aplicación son los campos que estudian la atmósfera o el subsuelo, como la climatología o la geología. Otra alternativa posible para el trabajo con este tipo de información volumétrica es el uso de sistemas de partículas, especialmente en el caso de información atmosférica.

El trabajo desarrollado en esta tesis doctoral se enfoca en el segundo y tercer puntos mencionados: la aplicación de imágenes *raster* de diferentes fuentes y naturalezas junto con los datos vectoriales 2D e información asociada procedente de SIG y CAD. En la literatura inglesa sobre el tema es habitual ver referencias a este tipo de información geográfica como **elementos no sobresalientes** (*non-protruding features*), diferenciándola claramente de los elementos sobresalientes de la superficie del terreno como los mencionados en el cuarto punto. Ejemplos de elementos sobresalientes pueden ser la vegetación, edificaciones u otras infraestructuras como puentes. Ejemplos típicos de elementos no sobresalientes son los ríos, carreteras, y todo tipo de delimitaciones de diferentes zonas del territorio, tanto físicas como políticas. En cualquier caso, el hecho de que un elemento se considere sobresaliente o no, depende de la escala de visualización y por tanto el mismo elemento real puede ser tratado de formas diferentes según el caso.

Existen numerosos trabajos que consideran el problema de la geometría del terreno. Dado que la arquitectura aquí descrita separa totalmente la gestión de estos tipos de información, se podría utilizar cualquier sistema de

gestión de geometría del terreno conjuntamente con la gestión de texturas y datos vectoriales combinados que se propone. Esta característica da una gran libertad al diseño del motor de geometría, que no se ve limitado por tener que ceñirse a una división espacial o a una organización jerárquica que coincida con la de las texturas utilizadas, como ocurre con la gran mayoría de las técnicas analizadas en el capítulo anterior.

Respecto al origen de los datos, éste puede ser muy variado, desde ficheros en disco local hasta datos recibidos a través de Internet desde un servidor remoto, o a través de una conexión punto a punto con un dispositivo de captura de datos, pasando por la información generada por *software* de simulación científica.

La información tratada en el sistema de visualización se puede clasificar en base a diferentes variables, de las cuales nos interesan principalmente las siguientes:

- Tipo de información (*raster* o vectorial)
- **Dinamismo** la información
- **Tamaño** de la información

La primera de las variables es de carácter binario en el sentido de que admite dos estados posibles: la información puede ser *raster* o vectorial. Ambos tipos de información se tratan de forma diferente, y por lo tanto se estudiarán por separado.

Las otras dos variables son continuas en el sentido matemático de que admiten infinidad de valores. De esta forma podemos clasificar la información dentro de un espacio bidimensional en función de su **dinamismo** y de su **tamaño**, tal y como se representa en la figura 3.1. En el origen tendremos la información estática y de tamaño mínimo. El eje X representa el dinamismo de la información y el eje Y su tamaño.

El dinamismo se corresponderá con la **frecuencia de actualización** (f) requerida por los datos que varíen en el tiempo, ya sea por recibirlos en tiempo real o por representar la evolución en el tiempo de datos históricos archivados. Este parámetro se mide en número de actualizaciones por segundo y está en relación con el parámetro inverso al cual denominaremos **tiempo de vida** (T) de los datos y que se medirá en segundos.

$$f = \frac{1}{T} \tag{3.1}$$

Aunque las variables descritas admiten múltiples combinaciones, destacan algunos escenarios típicos que se producen con bastante frecuencia. Por este motivo es importante describir estas situaciones habituales y tenerlas en cuenta a la hora de diseñar el sistema de visualización. Resultarán igualmente de gran utilidad a la hora de validar el sistema.

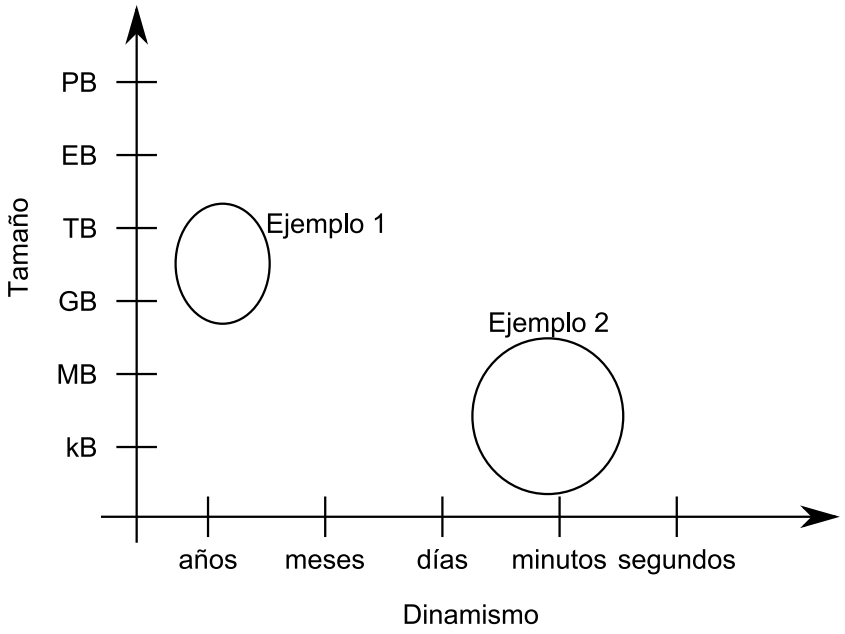


Figura 3.1: Análisis de las características de la información (dinamismo vs tamaño).

El primero de los escenarios es el caso de información *raster* de gran tamaño y estática (o cuasi estática). Un ejemplo muy habitual es el de imagen de satélite o fotografía aérea proyectada sobre el terreno. La actualización de esta información es poco frecuente, entre otros motivos porque su adquisición y procesado es un proceso lento y costoso. Su tiempo de vida será típicamente de meses o, más habitualmente, años. El tamaño de los datos es considerablemente elevado. Uno de los ejemplos con los que se ha trabajado en el proyecto SANTI y que se ha utilizado en el diseño, desarrollo y validación del sistema propuesto corresponde a una imagen aérea de 0.25 m/pixel de resolución de la Comunidad Gallega (unos 200x250 km aproximadamente) que ocupa unos 3 TBytes en disco (figura 3.2). Como muestra del tiempo de actualización de este tipo de información, se puede mencionar que el Plan Nacional de Ortofotografía Aérea (PNOA), del Instituto Geográfico Nacional prevé la cobertura de todo el territorio nacional cada dos años con ortofotos de 0,5 m/pixel de resolución [115].

Un segundo escenario planteado, antagonista del anterior, consiste en información vectorial 2D proveniente de un SIG, actualizada frecuentemente y de tamaño reducido (especialmente si se compara con el caso anterior).

El volumen de datos manejado en el caso de datos vectoriales puede ser muy variable, pero típicamente estará del orden de los MBytes. El tiempo de actualización puede ser muy variable según el tipo de información, desde la representación de infraestructuras o planes urbanísticos, con actualizaciones infrecuentes (del orden de meses o años) hasta los datos altamente dinámicos

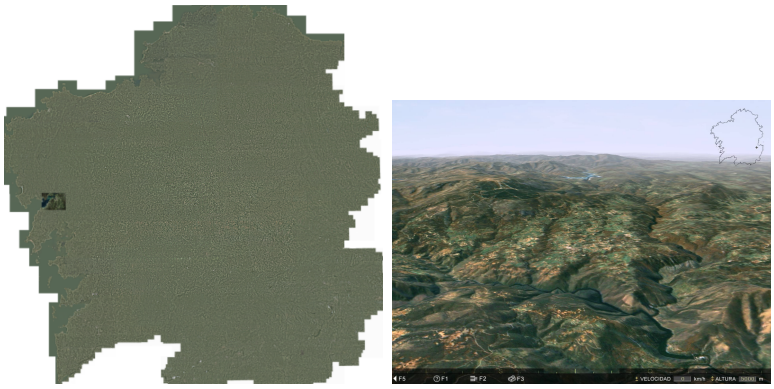


Figura 3.2: Ejemplo 1. Fotografía aérea de Galicia a 0,25 m/pixel mapeada sobre el modelo digital de elevación del terreno ($\sim$ 3 TBytes sin compresión).

como la información capturada en tiempo real de los sensores que miden la intensidad del tráfico, el seguimiento de vehículos mediante GPS o la información procedente de estaciones meteorológicas, con actualizaciones que pueden oscilar entre minutos y segundos. Un ejemplo de este tipo se ilustra en la figura 3.3.

Figura 3.3: Ejemplo 2. Red carreteras de Galicia con información de nivel de tráfico ($\sim$ 15 MBytes). Datos vectoriales aplicados sobre los datos *raster* (fotografía aérea).

Estos escenarios se tendrán en cuenta a la hora de validar el sistema y analizar el rendimiento, por considerarlos como los casos extremos dentro del abanico de posibilidades, además de tratarse de casos bastante habituales.

En conclusión, establecemos la definición del problema planteado en este trabajo como la gestión de esta información bidimensional, de características tan heterogéneas, sobre el modelo 3D del terreno para su visualización con una alta calidad al tiempo que se proporcionan los mecanismos para mantener un rendimiento interactivo adecuado.

En la siguiente sección se concretan los objetivos que se han establecido en cuanto a diferentes aspectos importantes. Finalmente se justifica la elección de la metodología empleada para abordar la resolución del problema planteado.

3.2. Objetivos

A la hora de definir los objetivos del sistema propuesto se han considerado los siguientes aspectos:

- Volumen de información
- Rendimiento
- Calidad de visualización
- Prestaciones
- Requisitos de *hardware*

3.2.1. Volumen de información

El volumen de información manejado viene determinado por dos factores: la extensión de terreno visualizada y la densidad o resolución de los datos en esa extensión.

En cuanto a la extensión geográfica, no se presupone ningún límite. El sistema debe poder trabajar correctamente con información de todo el planeta a cualquier nivel de detalle para el cual se disponga de fuentes de información. Esto presenta ciertos problemas de precisión numérica, que serán estudiados en el siguiente capítulo.

Se plantea un sistema que pueda gestionar conjuntos de datos aunque éstos desborden la capacidad del *hardware* utilizado, tanto en almacenamiento (a cualquier nivel) como en capacidad de cálculo. Por lo tanto, se utilizarán las técnicas de paginación y cachés adecuadas para gestionar eficientemente toda la información y la gestión de niveles de detalle para ajustar el detalle visualizado al necesario en cada momento, siempre dentro de los límites del *hardware*.

El sistema de visualización será por lo tanto completamente escalable en cuanto a la cantidad de información manejada, tanto en extensión como en densidad.

3.2.2. Rendimiento

El rendimiento es prioritario en el sistema propuesto. En caso de conflicto entre calidad y rendimiento prevalecerá siempre este último aspecto.

Como objetivo cuantificable de rendimiento se plantea un *frame rate* constante de 60 fotogramas por segundo (fps). Esto establece una cota máxima de de 16,66 ms al tiempo de *render*. Se proveerán los mecanismos necesarios para garantizar que este tiempo no sea superado en ningún momento.

La latencia de la visualización está condicionada por muchos aspectos externos al sistema propuesto, como la organización en procesos o hebras

(*threads*) de ejecución en paralelo del *scene graph*. Es habitual separar y ejecutar en paralelo las siguientes tareas:

- Actualización de la escena (frecuentemente denominada *app*[127] o *update*[18]). En esta fase se capturan las órdenes del usuario, se actualizan los estados de los elementos dinámicos de la escena, se ubica la cámara o las cámaras y se realizan otras tareas no correspondientes a la visualización, como la gestión del audio.

- Filtrado (*cull*). En esta fase se recorre el *scene graph* realizando tareas tales como descartar partes de de la escena por no entrar en la zona visible (*view frustum culling*) o por estar ocultas por otros elementos (*oclussion culling*), ordenación por propiedades de los objetos a visualizar para minimizar cambios de estado en el sistema gráfico, etc. Las misiones básicas de esta fase se pueden resumir en dos: optimizar el rendimiento ya sea por eliminar partes innecesarias o por estructurar la escena de forma más eficiente, y generar los conjuntos de elementos preparados para *renderizar*.

- Dibujado (*draw*). Esta fase se ejecuta en un contexto gráfico y básicamente consiste en enviar las órdenes de dibujado recopiladas y estructuradas en la fase anterior (*cull*) al API gráfico de bajo nivel utilizado (por ejemplo OpenGL o Direct3D). Estas órdenes serán recibidas por el controlador gráfico y procesadas en la GPU.

Existen otras tareas además de las anteriores que se pueden ejecutar en paralelo, como la paginación entre memoria principal y almacenamiento secundario o el cálculo de intersecciones. Como norma se paralelizará cualquier operación que suponga la lectura de datos desde una fuente de acceso lento (como un disco o un servidor a través de la red), en cuyo caso se realizan las cargas de forma asíncrona, o cualquier cálculo costoso (a no ser que sea absolutamente imprescindible disponer de él en cada fotograma). En general, aquellas operaciones que en caso de realizarse de manera síncrona provoquen un retraso en el *render* que desborde el tiempo disponible y haga caer el *frame rate*.

Además, las tareas listadas no son únicas, en muchos casos existirán procesos del mismo tipo independientes por cada vista de la escena, generalmente *cull* y *draw*, que son las fases que dependen completamente de la cámara utilizada.

Para establecer un objetivo cuantificable en cuanto a latencia, se asumirá que se cumplen las siguientes condiciones:

1. Se obtienen las órdenes del usuario en la fase *update*.

2. Se ejecutan la fases *cull* y *draw* en paralelo con la anterior.

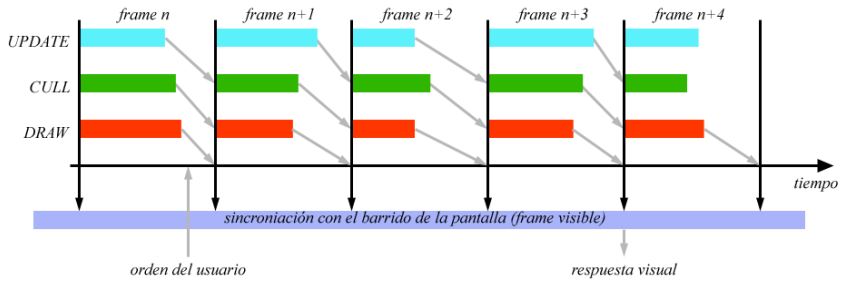


Figura 3.4: Latencia con un *pipeline* de tres fases en paralelo.

3. Se usa la técnica de *double buffering*[74] por lo que se mostrarán los fotogramas sincronizados con el barrido de la pantalla.

4. El tiempo de fotograma es de 16,66 ms, como se mencionó anteriormente, para obtener una frecuencia de 60 fps. Esto implica que ninguna de las fases de ejecución en paralelo podrá superar este tiempo.

Partiendo de estas premisas, se puede establecer una cota máxima para la latencia de cuatro veces el tiempo de *render* ($\sim$ 67 ms). Esto se ilustra en la figura 3.4, donde se muestra que la orden dada por el usuario en un fotograma n produce un resultado visible en el fotograma $n + 3$.

Todos los módulos del sistema propuesto implementan un control de tiempo para mantener la máxima calidad evitando superar el tiempo de *frame*, tal y como se detallará en el próximo capítulo. Estos mecanismos de control serán dinámicos, de forma que se adaptarán al estado de carga de la máquina. Este nivel de carga estará afectado por parámetros ajenos al sistema desarrollado en esta tesis, pero que no obstante serán tenidos en consideración a la hora de realizar los ajustes pertinentes para mantener el *frame rate* constante.

3.2.3. Calidad de la visualización

Los aspectos de calidad estarán enfrentados con el rendimiento. Es necesario, por lo tanto, buscar un equilibrio entre ambos, dando prioridad a éste, tal y como ya se ha expuesto.

Para poder evaluar la adecuada consecución de los objetivos de rendimiento recién planteados, debemos establecer unos mínimos criterios de calidad de la visualización. Se asume una resolución de 1280x1024 pixels, con 24 bits de profundidad de color.

El ángulo o campo de visión (FOV o *field of view*) puede afectar al rendimiento del *render*, puesto que la cantidad de información visible de la escena en un fotograma determinado es directamente proporcional a dicho parámetro. Sin embargo, por tratarse de un mecanismo de texturizado

independiente de la geometría, la técnica propuesta no se ve afectada negativamente por el FOV utilizado, por lo que en principio no se establece ningún límite al respecto.

Dentro de la resolución planteada, se aplicará el máximo detalle de textura necesario en cada momento (limitándose a la disponible en las fuentes de datos), lo cual implica que el tamaño de la porción cacheada de textura en el máximo nivel de detalle necesario debe superar la resolución de pantalla.

Otro aspecto fundamental para una buena calidad de visualización es eliminar los artefactos debidos al *aliasing* mediante un correcto filtrado, tanto de las texturas como de la geometría de la información vectorial. Esto es especialmente importante en la situación habitual en visualización de terreno en 3D de observarlo con una vista horizontal. Por lo tanto, se aplicarán técnicas de filtrado anisotrópico para mejorar la calidad de la visualización hasta donde el *hardware* disponible permita.

Tratándose de un sistema multirresolución, se plantea el objetivo de minimizar los efectos distractorios por cambios de niveles de detalle. Los cambios de nivel de detalle deberían ser inapreciables para el usuario, pero en aquellas ocasiones en que esto no es posible, se tratará al menos de minimizar su impacto en la calidad de la visualización.

3.2.4. Prestaciones

Independientemente de la calidad y el rendimiento, se plantean una serie de características que debe tener el sistema, enumeradas a continuación:

- Independencia de la gestión de geometría. En ningún caso el sistema de gestión de texturas virtuales debería condicionar la gestión del modelo geométrico de elevación del terreno, ni en la generación y estructura de los conjuntos de datos ni en la selección de niveles de detalle. Ambos sistemas deben ser completamente independientes en su tratamiento, tanto en los preprocesos *off-line* como durante la visualización interactiva.

- Implementable en *hardware* de bajo coste. Los trabajos previos en texturizado de grandes extensiones de terreno con alto nivel de detalle que cumplen el requisito anterior, exigen un *hardware* específico, accesible sólo para presupuestos elevados. El sistema desarrollado en este trabajo se puede implementar en ordenadores domésticos de gama media, como los utilizados para los videojuegos.

- Multitextura. La información SIG visualizada sobre el terreno se gestionará como una textura. Es imprescindible que el motor utilice una única unidad de texturizado para facilitar su combinación con otras texturas, ya sean gestionadas con este mismo sistema o con cualquier otro, en un único pase de *render*.

- Combinable con *shaders*. Aunque la información SIG se gestione como una textura, esta textura no debe limitarse a ser aplicada como información de color de manera inmediata. La información contenida en la textura puede ser utilizada desde *shaders* creados por los usuarios de la técnica para realizar cualquier efecto que se necesite en el futuro. Por ejemplo, puede resultar de utilidad el uso de *shaders* para *renderizar* las zonas cubiertas de agua, ya sean mar, ríos, lagos, embalses o cualquier otra, de forma más realista que con la fotografía estática. De esta forma se pueden simular el oleaje y las características de iluminación.

- Capas de información. Para una gestión eficiente de la enorme cantidad de información manejada por un SIG, ésta se agrupa en lo que se denominan capas. El sistema propuesto debe ser capaz de añadir, eliminar, mostrar, ocultar y variar el aspecto de estas capas de forma dinámica y ágil, para conseguir una buena respuesta interactiva. En ningún caso deberá ser necesario reiniciar la aplicación de visualización para poder incorporar nueva información procedente de cualquier fuente, incluidos servidores de datos geoespaciales a través de Internet o ficheros en disco.

- Además de la información vectorial habitual compuesta de puntos, líneas y polígonos, se debe poder incluir otro tipo de información dinámica como textos o iconos en la textura virtual. El hecho de gestionar esta información de manera dinámica facilita que se mejore la legibilidad de los textos independientemente de la vista del usuario (no necesita estar orientado hacia el norte, como ocurría en muchos de los sistemas estudiados).

- Integración en *scene graphs*. Todos los tipos de información heterogéneos que se manejan (ver figura 1.3) se combinan en la escena final mediante una estructura de **scene graph** [39].

  Desde el punto de vista del *scene graph*, el modelo del terreno, compuesto de la geometría junto con la información que se le aplique a modo de textura, se considera como un nodo. De esta forma se encapsula toda la gestión del terreno haciéndola transparente a la implementación concreta de *scene graph* utilizada en un determinado proyecto. Así pues, el sistema de gestión de terreno se puede utilizar dentro de cualquier motor de *scene graph*. Durante el desarrollo de esta tesis, se ha utilizado OpenSceneGraph [18] como motor de alto nivel donde se encapsula la visualización de terreno en un objeto dibujable del scene graph (clase `osg::Drawable`) que se incluirá en un nodo contenedor de objetos dibujables (clase `osg::Geode`).

  De la misma manera, la textura se encapsula dentro de un objeto textura (`osg::Texture`) con unas funciones de actualización (*callbacks*)

asociadas, de forma que se pueda aplicar a cualquier geometría del *scene graph*, no limitándose al terreno gestionado por un motor específico.

Esto proporciona una gran versatilidad al sistema, permitiendo combinarlo con cualquier tipo de modelo 3D y con todas las características soportadas en el motor del *scene graph* utilizado.

- Nivel de detalle *raster* heterogéneo. Es imprescindible que se pueda manejar información *raster* con detalle heterogéneo de forma perfectamente integrada. En cada zona se mostrará el nivel de detalle necesario dentro del disponible en las fuentes de datos. Esto es similar a lo que en clipmapping[144] se denominaban *insets* de alta resolución.

- Información dinámica. La información visualizada será variable en grado de dinamismo. El diseño del sistema tiene en cuenta este aspecto para gestionar de forma eficiente el conjunto de datos dinámicos con una frecuencia y condiciones de cambio muy variables. En ningún caso se debe requerir reiniciar la aplicación de visualización para poder incorporar nuevos datos o actualizar los existentes, como por ejemplo la imagen de satélite o la fotografía aérea del terreno. Estos cambios se podrán realizar siempre "en caliente" durante cualquier visualización sin interferir en su correcto funcionamiento.

- Múltiples vistas. El sistema propuesto debe soportar múltiples vistas de la misma escena. Esta capacidad es interesante para visualización estereoscópica, sistemas multipantalla con diferentes grados de "inmersividad", desde *videowalls* hasta *CAVEs*, *domes* o simuladores de vuelo con diferentes vistas desde la cabina. Para optimizar el rendimiento y minimizar los recursos necesarios, los datos independientes de la vista deben estar compartidos. En el diseño de la jerarquía de cachés se tiene en cuenta este aspecto de forma que se facilita la reutilización de los datos desde múltiples cámaras simultáneamente.

3.2.5. Requisitos de *hardware*

El diseño de un sistema de este tipo, integrando información SIG en un modelo 3D de terreno está condicionado por dos tipos de restricciones: las restricciones de *hardware* y los requisitos de los usuarios [102]. Como estos requisitos ya se han considerado en los puntos anteriores, únicamente queda establecer cuáles serán los requisitos en cuanto a *hardware*.

Puesto que los principales objetivos son la flexibilidad, calidad y rendimiento, y la complejidad del sistema es considerable, los requisitos de *hardware* no son tan reducidos como en los sistemas precursores [137]. Para la implementación de las técnicas propuestas se necesita una GPU con soporte

de *vertex* y *fragment shaders*. En concreto se necesita soporte para el *Shader model 3.0*.

El diseño del sistema es muy flexible y admite diferentes configuraciones, pudiendo adaptarse en gran medida a los recursos disponibles. Sin embargo, para obtener una buena calidad y rendimiento, aprovechando al máximo todas las prestaciones del sistema, es recomendable utilizar GPUs de última generación, con soporte para el *Shader model 4.0*, como la serie 8 de NVIDIA, cuyos precios están a nivel doméstico.

Una importante restricción provocada por el *hardware* utilizado es la limitación en la precisión numérica con la que se trabaja. Los sistemas gráficos de consumo actuales trabajan con valores flotantes de 32 bits (ANSI/IEEE Standard 754-1985), lo cual provoca errores de precisión cuando se trabaja a escalas muy diversas, que van desde una vista global del planeta hasta detalles centimétricos del terreno. El diseño se ha realizado teniendo en cuenta esta limitación del *hardware*.

Para la validación del sistema se ha utilizado principalmente un equipo doméstico de gama media: Dell Inspiron 9150, con CPU Pentium D 950 3 GHz (doble núcleo), 2 GBytes de RAM y tarjeta gráfica NVIDIA GeForce 8800 GTS (PCI-express) con 320 MBytes de VRAM . Aunque esta ha sido principalmente la máquina de desarrollo, también se han realizado pruebas y analizado el rendimiento con otras configuraciones, tal y como se describe en el capítulo de resultados.

3.3. Metodología

Como ya se ha descrito en profundidad en las secciones 2.7 y 2.8 del capítulo anterior, existen varias aproximaciones para resolver el problema planteado. En este trabajo se propone un sistema basado en técnicas de texturizado para integrar la información *raster* y vectorial 2D sobre el terreno 3D.

El sistema de gestión de texturas de tamaño virtualmente ilimitado sigue la filosofía de la "textura de terreno global" de Michael Cosman [55] y más concretamente la estructura de clipmap planteada posteriormente por Tanner [144]. Esto soluciona completamente los problemas de acoplamiento con la geometría de los que adolecen la gran mayoría de técnicas de texturizado de terreno (o de gestión de texturas de gran tamaño), al tiempo que, a diferencia de los anteriormente mencionados, no se necesita un *hardware* específico ni excesivamente caro.

La integración de los datos vectoriales 2D sigue la idea planteada por Kersting [96] de generación de textura bajo demanda, frente a otros sistemas que defienden el uso de geometría [146, 33, 132, 134].

Se presentan soluciones a los problemas planteados por los detractores de la representación mediante texturas de la información SIG vectorial 2D, al

tiempo que se ofrecen características no disponibles en los sistemas basados en geometría, y se evitan los numerosos problemas que presentan, tal y como ya se ha descrito anteriormente.

Capítulo 4

Desarrollo

En este capítulo se detallan los aspectos más relevantes de la arquitectura *software* y las técnicas desarrolladas en este trabajo.

Se comienza en la sección 4.1 con una descripción global de la **arquitectura del sistema**, cómo ha sido estructurado, las interrelaciones entre los diferentes módulos, las capas de abstracción definidas, las interfaces entre dichos módulos y la interfaz pública de desarrollo de aplicaciones (API).

El motor de texturas virtuales dinámicas se ha denominado **GeoTextura** y constituye el componente principal del núcleo del sistema desarrollado (figura 4.1). Los contenidos de este capítulo girarán en torno a este componente, con algunos aspectos ampliados y completados en los apéndices.

En la sección 4.2 se describen ciertas **capas de abstracción** que proporcionan **interfaces unificadas** para el acceso a las GeoTexturas, combinaciones de ellas o con otros motores de texturizado diferentes. Estas capas de abstracción facilitan el uso de diferentes motores de texturizado o combinación de ellos de forma transparente al usuario o aplicación. Las interfaces se detallan posteriormente en el apéndice A.

La sección 4.3 se centra en la estructura del motor de texturizado (GeoTextura). Este componente se basa en una **caché de dos niveles** (figura 4.4), donde el primer nivel (L1 o **CacheTRAM**) almacena un subconjunto de los datos de textura en la memoria de vídeo y el segundo nivel (L2 o **CacheRAM**) almacena un subconjunto de los datos en memoria principal. Los apéndices E, F y G complementan la descripción de estos componentes.

El funcionamiento de una GeoTextura se basa en dos procesos clave: el ***render*** y la **actualización**. En esta misma sección se describirá también la fase de *render*, detallada posteriormente en el apéndice B.

Las dos siguientes secciones tratan el proceso de actualización de una GeoTextura. La **actualización síncrona** de la CacheTRAM, se describe en la sección 4.4 y la **actualización asíncrona** de CacheRAM, en la sección 4.5.

En la sección 4.6 se trata el caso especial de la *rasterización* de **informa-**

ción de origen vectorial, que tiene una problemática y un tratamiento, en cuanto al proceso de actualización, completamente diferente a las texturas cuyo formato en origen es de tipo *raster*.

Finalmente, la sección 4.7 y el apéndice D describen los problemas inherentes a la gestión de texturas de tamaño virtualmente ilimitado, que pueden superar las capacidades de direccionamiento del *hardware*, y cómo se han resuelto dichos problemas.

4.1. Arquitectura general

Partiendo de la visión global descrita en los capítulos anteriores, comenzamos por definir el dominio de este estudio y acotar el problema planteado dentro de su entorno. Tal y como se ha descrito en la introducción, el trabajo se integra dentro de un sistema interactivo de visualización de terreno 3D e información geográfica, cuya arquitectura general se ilustra en la figura 1.3. Nos limitaremos únicamente a la aplicación de información gráfica bidimensional (tanto *raster* como vectorial) sobre el modelo digital tridimensional de terreno mediante técnicas de mapeado de textura.

El sistema global de visualización de terreno 3D e información geográfica está basado en un sistema de *scene graph* (OSG en la versión actual). La gestión del modelo digital de elevación del terreno se realiza mediante un módulo específico para la visualización eficiente de geometría 3D de terreno. Este módulo está integrado dentro del *scene graph* como un objeto dibujable (`osg::Drawable`). El motor de geometría es el principal cliente del sistema de texturizado propuesto, aunque éste estará disponible para otros elementos del *scene graph*, tal y como se describe más adelante[1].

Típicamente, los sistemas de visualización de terreno se componen de dos partes fundamentales: el subsistema de procesamiento de la información a visualizar y el subsistema de visualización en tiempo real. El primero de los dos subsistemas trata la información que se mostrará al usuario adecuándola para su eficiente gestión en tiempo real, que será realizada por el segundo. Ambos aspectos del sistema global afectan al motor de texturizado desarrollado en este trabajo, por lo que deben ser tenidos en cuenta.

Estos tratamientos de la información del terreno suelen ser costosos en tiempo y uso de memoria, por lo que habitualmente se realizan en una fase previa a la visualización (preprocesamiento *off-line*). Es decir, se realiza un preproceso sobre los datos originales para generar una base de datos optimizada para su visualización en tiempo real. Sin embargo, como el sistema objeto de este estudio se basa en la visualización de información dinámica, en la mayoría de los casos esta información se obtiene en tiempo real des-

[1] Por ejemplo, es habitual utilizar la imagen aérea para mapear, además del modelo de terreno, las cubiertas de los edificios que se reconstruyen en 3D, que se mostrarán así con su aspecto real y coherente con el resto del terreno.

de su origen, de forma que el preprocesamiento no queda más remedio que realizarlo "al vuelo".

Una vez descrito el problema que se va a abordar, comenzamos por estructurar la solución propuesta en diferentes módulos y definir su interfaz con el exterior del sistema. La figura 4.1 ilustra la arquitectura global, que se explica a continuación.

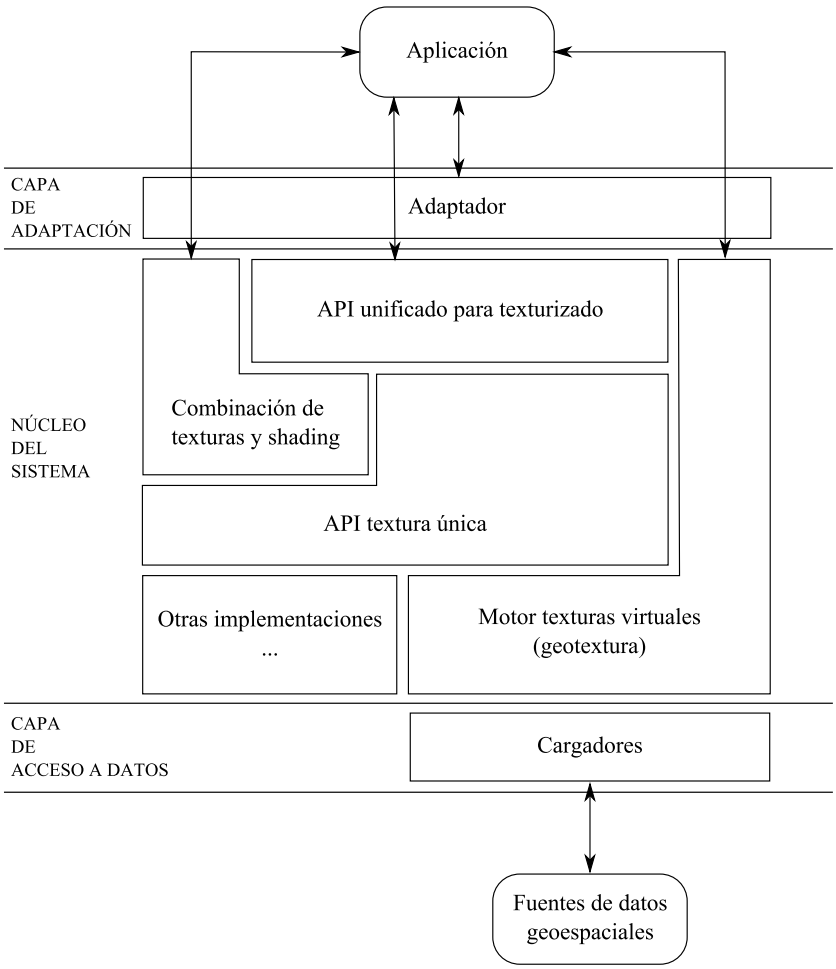


Figura 4.1: Estructura general e interfaces del sistema propuesto.

Se divide el sistema en tres capas: el núcleo del sistema, una capa de acceso a datos georreferenciados, y una capa de adaptación opcional.

4.1.1. Núcleo del sistema de texturizado dinámico

La arquitectura del núcleo presenta interfaces a varios niveles. En primer lugar, para una mayor versatilidad y comodidad, se ofrece una interfaz abstracta genérica de texturizado. Esta interfaz unificada ofrece a la aplicación

cliente toda la funcionalidad necesaria para aplicar la textura de forma similar a cómo se haría en una API gráfica de bajo nivel como OpenGL o DirectX. Existirán al menos una llamada para aplicar la textura y otras dos para almacenar y restaurar respectivamente los estados del sistema gráfico afectados por el motor de texturizado.

Mediante este mecanismo unificado se podrá utilizar cualquier textura de la misma forma sin conocer sus detalles internos, pudiendo tratarse de una textura normal, una textura virtual, o un *shader* que combine diversas texturas virtuales junto con otros efectos programados en la GPU.

Para poder realizar estas combinaciones de texturas y efectos, se dispone de un sistema encargado de la combinación de texturas y *shading*. Este módulo no sólo contiene la interfaz sino el propio sistema de combinación de texturas simples[2].

Estas texturas simples están disponibles a través de la interfaz abstracta que se indica en la imagen como "API textura única". Esta interfaz abstracta permite que en la arquitectura que se plantea coexistan diferentes sistemas de texturizado, no limitándose únicamente al desarrollado en esta tesis doctoral. Por lo tanto, en el futuro se podrán integrar sistemas alternativos de texturizado de forma transparente a las aplicaciones cliente.

Por último, la pieza principal del sistema, que implementa las técnicas desarrolladas en este trabajo es el motor de texturas virtuales que hemos denominado **GeoTexturas**. Este sistema ofrece también una API para poder realizar las tareas específicas del motor de GeoTexturas, en el que se profundizará a lo largo de este capítulo.

Durante el *render* se utilizará principalmente la API unificada. Las otras interfaces se utilizarán principalmente en la fase de inicialización para configurar los esquemas de texturas y efectos que se vayan a utilizar. También puede ser necesario acceder a la interfaz de los subsistemas que gestionen las texturas virtuales y/o dinámicas, para poder realizar tareas específicas como cambios de configuración, fuentes de datos, etc.

4.1.2. Capa de adaptación

El sistema se puede utilizar directamente y con todas sus prestaciones a través de las interfaces anteriormente mencionadas, presentes en el núcleo. Sin embargo, cuando se utiliza en aplicaciones basadas en algún tipo de *scene graph*, resulta más cómodo y adecuado integrarlo dentro del mismo. Esto se realiza en una capa aislada que encapsula el motor de texturizado para el *scene graph* correspondiente.

Lo habitual es construir una subclase de las texturas del *scene graph*, de forma que las texturas virtuales dinámicas se puedan aplicar de forma

[2]Donde por textura simple nos referimos a una única textura, en contraposición a una multitextura, lo cual de ninguna manera quiere decir que las texturas ni su gestión sean sencillas.

transparente a cualquier modelo presente en la escena.

Respecto a las tareas de actualización mencionadas que se realizan mediante la API del sistema de texturas virtuales dinámicas, resulta útil integrarlas en *callbacks* de diferentes nodos u objetos del *scene graph*.

Estos callbacks pueden estar asociados a diversos elementos. Uno de los más habituales son los *callbacks* de cámaras. De esta forma se facilita mantener actualizadas cachés independientes para cada vista de la escena.

Otra alternativa es realizar las funciones de actualización una vez por fotograma, en caso de tener una única vista o que las cachés de la textura virtual se compartan entre las diferentes vistas.

En ocasiones la geometría del terreno se encapsula dentro de un nodo o un dibujable del *scene graph*. Esta aproximación es la que se utiliza actualmente en el proyecto SANTI [83]. En estos casos, las tareas de actualización de la textura aplicada a ese nodo pueden ser invocadas desde *callbacks* asociados al elemento del terreno.

En este trabajo se ha diseñado e implementado una capa de adaptación del sistema de texturizado para el motor de geometría de terreno del proyecto SANTI y para el *scene graph* OSG [18], de forma que se puede aplicar la textura al terreno o a cualquier otro modelo 3D.

4.1.3. Capa de acceso a datos

Para facilitar el acceso a diferentes fuentes de datos (que pueden ser muy heterogéneas en cuanto a características de los datos, sistemas de referencia, métodos de acceso, etc), el sistema propuesto dispone de los mecanismos de abstracción que encapsulan las tareas de carga (o generación) de los datos haciéndolas transparentes al subsistema de texturizado que solicita esos datos. Además de encapsular los métodos de acceso a los datos, el sistema de carga de información geoespacial se puede utilizar para realizar procesos sobre ellos como conversiones de formato, reproyección, descompresión, etc. De esta forma se puede solicitar información de la misma clase a diferentes fuentes aunque éstas los sirvan utilizando diferentes lenguajes, formatos, tipos de datos, *data* geográficos de referencia, etc.

Cada fuente de datos que se utilice en el sistema tendrá un cargador asociado. La estructura de un cargador típico se ilustra en la figura 4.2. Los cargadores serán diferentes según el tipo de fuente de datos a la que se acceda. Estas incluyen datos *raster* o vectoriales, estáticos o dinámicos y locales o remotos.

Algunos ejemplos de cargadores implementados en este trabajo son los siguientes:

- Imagen *raster* almacenada en ficheros en disco, con los niveles de la pirámide prefiltrados y estructurada para optimizar el rendimiento y la calidad.

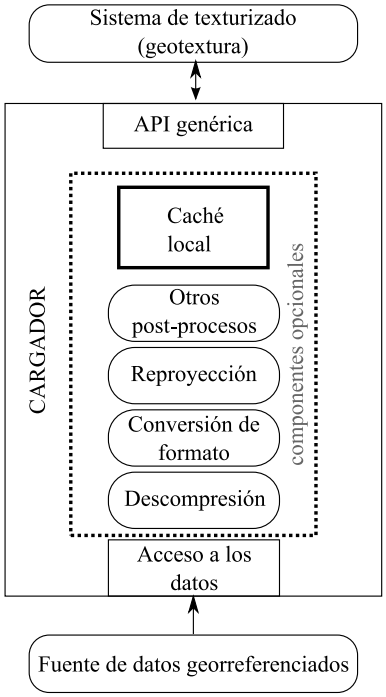


Figura 4.2: Esquema genérico de acceso a los datos

- Imagen *raster* similar a la anterior pero sin utilizar sistema de ficheros, accediendo directamente al dispositivo, para maximizar la transferencia desde disco, reduciendo la sobrecarga (*overhead*) asociada al sistema de ficheros.
- Imagen *raster* obtenida a partir de ficheros en formato de imagen georreferenciada, como ECW o GeoTIFF. Para el acceso a estos datos se ha utilizado la librería GDAL [8], por lo que se puede manejar cualquier formato soportado por dicha librería.
- Imagen *raster* o datos vectoriales suministrados a través de la red mediante servicios como WMS, TMS, WFS, WCS, ...
- Datos vectoriales procedentes de ficheros en el formato *shape* (.shp) de ESRI [7].
- Datos vectoriales procedentes de un *scene graph* genérico como OSG.
- Datos vectoriales procedentes de un *scene graph* específico para información SIG vectorial.

En todos los casos, los datos pueden ser tanto estáticos como dinámicos, por lo que la información lleva un tiempo de validez asociado y se volverá a solicitar a la fuente una vez expirado dicho tiempo.

En algunos casos, el cargador asociado a una fuente de datos puede almacenar en una caché interna la información obtenida de esa fuente. Esto es especialmente importante en el caso de acceso a servidores remotos, como realiza por ejemplo el cargador de WMS. La información obtenida se almacena en el disco local para evitar, en futuras peticiones de esos mismos datos, sobrecargar la red y el servidor remoto, al tiempo que se mejora la velocidad de acceso.

4.1.4. Entorno de desarrollo

Antes de comenzar el desarrollo del sistema de texturizado, se define el entorno sobre el que se trabajará. Uno de los principales objetivos planteados que han condicionado la elección es la portabilidad del sistema, por lo que se han evitado plataformas o *software* privativo, escogiendo en su lugar soluciones abiertas y estandarizadas en la medida de lo posible.

El diseño de *software* realizado sigue el paradigma de orientación a objetos. Se han utilizado también diversos patrones de diseño [75] para resolver problemas conocidos. Como lenguaje de modelado para la fase de diseño del *software* se ha utilizado UML [82].

La implementación se ha realizado en lenguaje ISO/ANSI C/C++ y debería funcionar en cualquiera de las plataformas (tanto *hardware* como *software*) más extendidas y en concreto en aquellas que soporten la familia de estándares POSIX.

Las operaciones gráficas se realizan a través de la API OpenGL 2.1 [135] (aunque el sistema seriá fácilmente portable a otras, como DirectX). Para la definición de *shaders*, incluidos los que realizan la gestión de las texturas virtuales, se utiliza el lenguaje GLSL [97, 128], perfectamente integrado con la API de OpenGL y que además permite la opción de compilarlos y enlazarlos en tiempo de ejecución. Esto último ha sido un punto clave en el diseño del sistema, puesto que ofrece enormes posibilidades en cuanto a cambios de comportamiento durante la ejecución del sistema sin aumentar por ello la complejidad ni el tamaño de los *shaders* (que ya de por sí son ambos bastante elevados).

Como formato para almacenar los datos de configuración del sistema se ha utilizado el estándar XML del World Wide Web Consortium (W3C). Se han diseñado esquemas XML para definir la configuración de los diferentes objetos utilizados.

Algunos componentes opcionales utilizan OSG [18] sobre la API gráfica de bajo nivel (OpenGL) para gestionar la estructura de *scene graph*. El sistema se integra perfectamente dentro de la escena de OSG, aunque puede ser utilizado de forma aislada en cualquier aplicación basada en OpenGL.

4.2. Interfaz del sistema (MotorTextura)

La interfaz de aplicación del sistema desarrollado unifica el acceso a las texturas, ofreciendo un método de acceso común para cualquier tipo de textura (virtual o no, estática o dinámica) o combinación de texturas en cualquier efecto, sin limitación más allá de las propias del *hardware* gráfico utilizado. En la figura 4.3 se muestra el diagrama de clases correspondiente a la interfaz pública del sistema de texturizado.

Esta capa de abstracción se implementa en la clase que se ha denominado **MotorTextura**. Podemos entender, por tanto, que un objeto MotorTextura puede ser una única textura de cualquier tipo o cualquier combinación de una o más texturas con cualquier efecto programado a través de *shaders*.

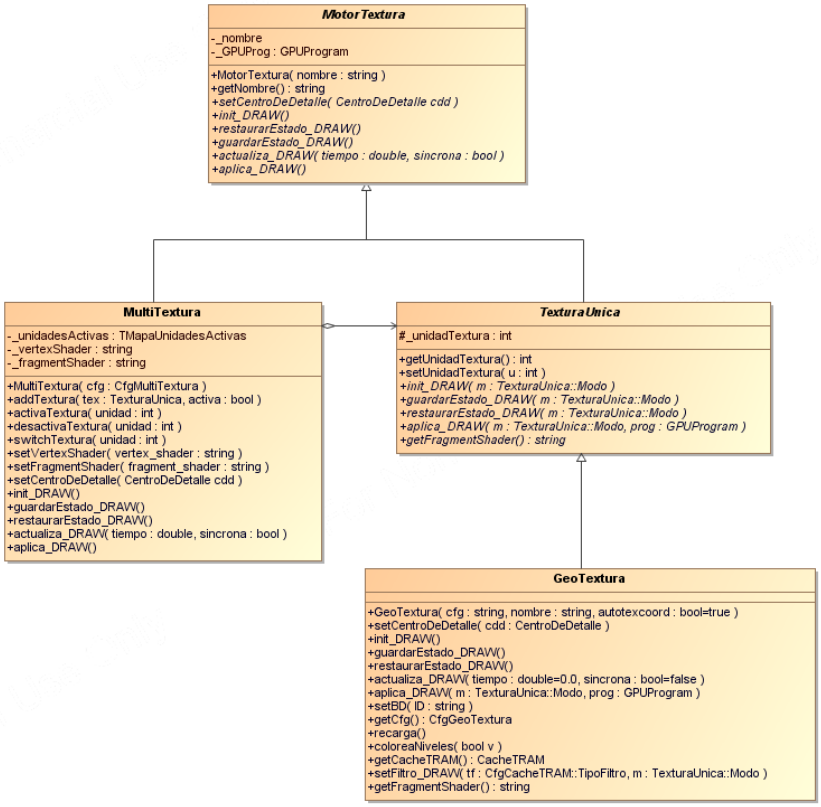


Figura 4.3: Interfaz de texturizado

MotorTextura es una clase abstracta que define la interfaz que deben implementar las subclases. Se han definido dos subclases, denominadas **MultiTextura** y **TexturaUnica**.

Existen algunos casos en los que se podría hacer necesario el acceso a las interfaces de las clases que están por debajo de la interfaz unificada de texturizado (MotorTextura). En primer lugar es necesario para la configu-

ración de las texturas y *shaders* a utilizar, lo cual sólo se realiza en la fase de inicialización del sistema.

Es importante destacar que parte de esta inicialización se debe realizar dentro del contexto gráfico en que se van a utilizar las texturas. En general, para la nomenclatura de los métodos de los objetos que deben ser invocados dentro de un contexto gráfico válido se ha utilizado el criterio de añadir el sufijo "_DRAW".

Estas tareas de configuración e inicialización se pueden encapsular en ciertos métodos de la interfaz MotorTextura, mediante el uso de ficheros de configuración que la subclase concreta interpretará para que los objetos se autoconfiguren. Para los ficheros de configuración se han utilizado documentos en formato XML.

La inicialización a partir de un fichero de configuración se combina con el uso del patrón "fábrica abstracta" [75] y un mecanismo de registro automático de prototipos para la construcción y configuración de objetos MotorTextura, ya sean MultiTextura o cualquier implementación de TexturaUnica. Esto permite utilizar de forma transparente y fácilmente extensible el sistema a través de esta interfaz unificada. La configuración específica de cada elemento del sistema se establecerá de esta manera en tiempo de carga a partir de su fichero de configuración.

En el caso de configuración dinámica y sobre todo de la modificación de propiedades específicas de los elementos componentes del motor de texturizado, puede ser más adecuado (o en algunos casos imprescindible) acceder directamente a la interfaz de las subclases.

Existen otros casos donde hace falta acceso a la interfaz de determinados componentes internos del sistema y que no se puede resolver de la manera recién descrita. Esto es así porque no se trata de la fase de inicialización y configuración del sistema sino que son tareas que se deben realizar de forma continua durante el *render*.

Ejemplos de este tipo de tareas son la actualización de las cachés, incluyendo el determinar la posición de la zona de interés y la paginación para descartar los datos invalidados y cargar en su lugar los nuevos datos requeridos para esa zona de interés.

En estos casos existen dos alternativas:

1. Incluir estas tareas necesarias para determinados tipos de subclases en la interfaz unificada. Esto sobrecarga dicha interfaz con métodos que algunas subclases no necesitan en absoluto.

2. Permitir el acceso a la interfaz de las subclases para realizar estas tareas. Al tratarse de tareas periódicas, se realizarán las llamadas dentro del bucle de *render* desde los lugares precisos (en algunos casos dentro del contexto gráfico). En caso de utilizar un *scene graph*, es habitual realizar estas tareas desde *callbacks*, tal y como se ha mencionado previamente.

Ambas alternativas tienen sus ventajas e inconvenientes. En el segundo caso se rompe un poco la filosofía de la interfaz unificada, pero la primera solución resulta poco extensible, pues la creación de nuevas clases contaminaría esa interfaz unificada añadiendo nuevas operaciones específicas de ciertas subclases. Por este motivo, se ha optado por trasladar a la interfaz unificada únicamente las tareas que se han considerado básicas e imprescindibles para la gestión de cualquier textura virtual dinámica: establecimiento de la zona de interés y actualización de la caché de textura en memoria de vídeo. De esta forma la sobrecarga es mínima y aunque esas dos llamadas no tendrán aplicación en el caso de texturas normales, ésto no supone ninguna diferencia notable en el rendimiento de la visualización.

Otra ventaja del planteamiento propuesto es que se unifica la gestión de la actualización de texturas virtuales combinadas mediante un objeto MultiTextura, sin que por ello se pierda la posibilidad de acceder directamente a la API de las subclases de TexturaUnica para operaciones especiales, como establecer diferentes zonas de interés a cada textura.

Un detalle a destacar en la interfaz planteada es que la zona de interés, aunque en la técnica desarrollada en este trabajo consiste en una única posición en 2D sobre el terreno (centro de detalle), se encapsula en un tipo abstracto, de forma que se podrían utilizar otras aproximaciones diferentes (por ejemplo, establecer diferentes centros de detalle para cada nivel o establecer áreas de interés en lugar de una posición central).

En el apéndice A se detallan las interfaces de las clases que componen los mecanismos de uso del motor de textura.

4.3. Textura virtual dinámica (GeoTextura)

El corazón del sistema de visualización en tiempo real de información geográfica 2D sobre el terreno 3D es el motor de gestión de texturas virtuales dinámicas que se ha denominado **GeoTextura**.

Este motor encaja dentro de la arquitectura y las interfaces descritas en los apartados anteriores como una implementación concreta de TexturaUnica, que podría coexistir con otras implementaciones utilizándose de manera simultánea todas ellas.

La arquitectura del motor de GeoTextura, ilustrada en la figura 4.4, se estructura como una cadena de cuatro bloques que constituyen una caché a dos niveles.

El primer nivel corresponde a la memoria de texturas del sistema gráfico, que denominaremos TRAM[3]. Este primer nivel de caché en TRAM está

[3]En los sistemas utilizados en este desarrollo, y en la mayoría de los sistemas gráficos en la actualidad, se utiliza la memoria de vídeo común a todo el sistema gráfico, denominada habitualmente VRAM, para almacenar las texturas al igual que muchos otros recursos (*frame buffer*, *vertex buffer objects* (VBOs), *pixel buffer objects* (PBOs), etc.). En este

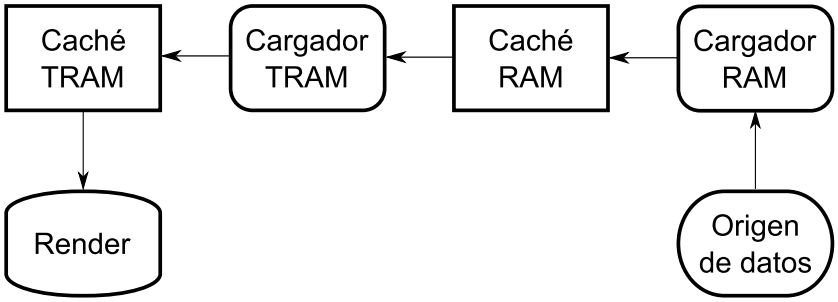


Figura 4.4: Arquitectura global del sistema de gestión de texturas virtuales dinámicas (GeoTextura).

definido en la superclase abstracta **CacheTRAM**, que se apoya en un módulo encargado de realizar las cargas de texturas a dicha caché. El cargador que alimenta a CacheTRAM se define en la superclase abstracta **CargadorTRAM**, y a su vez solicita la información al segundo nivel de caché, que almacena la información en la memoria principal del ordenador.

Esta caché de segundo nivel se define en la superclase abstracta **CacheRAM**. Puesto que el acceso a los orígenes de datos, ya sea disco, red o cualquier otro dispositivo, será el cuello de botella en el flujo de datos de la cadena descrita, este componente deberá realizar una carga predictiva de forma que se minimizen los fallos de caché y por tanto se optimize la velocidad de transferencia efectiva en la cadena. Por este motivo se podrán tener en cuenta parámetros como la dirección y velocidad de avance de la cámara a la hora de determinar qué zonas serán cargadas. Se realizarán las cargas de forma asíncrona, en un proceso independiente, para evitar afectar al rendimiento de la visualización y mantener una frecuencia de refresco constante.

La última pieza de la cadena es el acceso al origen de los datos geoespaciales. Esta operación se define en la superclase abstracta **CargadorRAM**, y en la arquitectura global de la figura 4.1 corresponde a la capa de acceso a datos, descrita en la sección 4.1.3.

Las tres primeras etapas corresponden al núcleo del sistema de la figura 4.1 y serán descritas en las siguientes secciones de este capítulo y en los apéndices B E y F. Los detalles de diseño de la cuarta se describirán en la sección G.

El tándem de componentes CacheRAM/CargadorRAM no tienen por qué limitarse a alimentar una simple caché en RAM, sino que pueden tomar un papel activo en la generación de textura bajo demanda a partir de información vectorial. En la sección 4.6 se describe el funcionamiento de la generación de textura a partir de datos vectoriales.

---

texto se ha utilizado el término TRAM por ser más específico, aunque en el contexto actual podría sustituirse por VRAM.

Durante el diseño del sistema se ha establecido la premisa de que el tamaño de la textura virtual sea una potencia entera de dos en ambos ejes. Esta ha sido una restricción habitual en los motores de texturizado por *hardware*, especialmente con el uso de *mipmaps*, puesto que permite acelerar el proceso para conseguir un buen rendimiento en el *render*, aunque en las últimas generaciones de *hardware* han salido extensiones que permiten evitar esta restricción [35]. No obstante, la restricción indicada corresponde al espacio virtual direccionado por el sistema, que no tiene por qué contener información en todas las zonas de todos sus niveles, por lo que la restricción no debería suponer ninguna limitación para el cliente o usuario.

La interfaz de la clase GeoTextura se detalla en el apéndice A. En los siguientes apartados de este capítulo se detallan y justifican las principales decisiones de diseño tomadas durante el desarrollo del sistema. Para facilitar la lectura, se han separado en los apéndices finales de la memoria algunos aspectos técnicos del sistema desarrollados en mayor detalle.

4.3.1. Coordenadas de textura

El vínculo entre el modelo geométrico del terreno y la información bidimensional que se aplica sobre éste en forma de textura se establece a través de las coordenadas de textura, que asocian a cada vértice de la geometría la zona correspondiente en el espacio textura.

Las coordenadas de textura se pueden suministrar de dos formas diferentes. En primer lugar se pueden enviar las coordenadas asociadas a los vértices en las órdenes de dibujado (`glTexCoord2f`), ya sea en modo directo, mediante *display lists*, VBOs, o cualquier otro mecanismo disponible.

La segunda alternativa consiste en calcular estas coordenadas de textura automáticamente en un programa por vértice ejecutado en la GPU. Conociendo el área geográfica que cubre la textura, se pueden obtener las coordenadas de textura de cada vértice a partir de su posición de manera trivial. El caso más complicado será aquel en que la geometría y la textura estén en sistemas de coordenadas distintos, puesto que habrá que realizar la transformación entre los diferentes espacios.

La primera opción puede reducir carga en aquellos casos en que el programa por vértice sea el cuello de botella del sistema o tenga una carga importante. Sin embargo, la segunda evita almacenar en memoria de vídeo las coordenadas de textura precalculadas y/o transferirlas por el bus desde memoria principal hasta el sistema gráfico. Cuando se utilicen sistemas de geometría dinámica, por ejemplo niveles de detalle adaptativos, no es posible precalcular las coordenadas de textura para los vértices, puesto que éstos varían su posición continuamente. En estos casos no queda más remedio que calcular en cada fotograma las coordenadas de textura correctas en función de la posición del vértice en ese preciso instante.

El motor de texturas virtuales dinámicas desarrollado en este trabajo

permite ambas posibilidades. En el constructor de la clase GeoTextura se pasa un parámetro `autotexcoord` que indica si se desea la generación automática de las coordenadas de textura dentro del programa por vértice o si por el contrario se utilizan las coordenadas suministradas junto con la posición de los vértices.

En el primer caso, el programa por vértice realiza el cálculo de las coordenadas de textura en función de la posición del vértice, como se muestra en la figura 4.5. La GeoTextura se encarga de pasar como parámetro de tipo *uniform* el área cubierta por la textura en el sistema de coordenadas geográficas utilizado.

```
uniform vec4 areaTex; // (xmin, ymin, ancho, alto)
uniform sampler3D tex;
#define uniTex 0

void main()
{
    // Aplica al vértice las transformaciones de OpenGL
    gl_Position = ftransform();
    // Cálculo automático de las coordenadas de textura
    gl_TexCoord[uniTex] = vec4(
        (gl_Vertex[0] - areaTex[0]) / areaTex[2],
        (gl_Vertex[1] - areaTex[1]) / areaTex[3],
        0.0, 1.0 );
}
```

Figura 4.5: Programa por vértice para la autogeneración de las coordenadas de textura.

En el segundo caso, el programa por vértice simplemente transfiere esas coordenadas de textura al programa por *fragment*. En la figura 4.6 se muestra el código por omisión del *fragment shader*.

```
uniform sampler3D tex;
#define uniTex 0

void main()
{
    // Aplica al vértice las transformaciones de OpenGL
    gl_Position = ftransform();
    gl_TexCoord[uniTex] = gl_MultiTexCoord0;
}
```

Figura 4.6: Programa por vértice para el paso de coordenadas de textura junto con la información de los vértices.

Además de estos dos comportamientos prediseñados, el usuario tiene la libertad de suministrar el código fuente en lenguaje GLSL para el programa por vértice que desee utilizar. Únicamente deberá asegurarse de que dicho programa suministra las coordenadas de textura mediante el parámetro *varying* de GLSL `gl_TexCoord` en el sistema de coordenadas adecuado.

4.3.2. CacheTRAM

La caché en TRAM es el elemento más importante del motor de texturizado, puesto que es el único que se utilizará en el proceso de *render*. El resto de los componentes de la cadena tienen la misión de mantener actualizados los contenidos de esta caché para conseguir la máxima calidad en el *render* sin afectar al rendimiento.

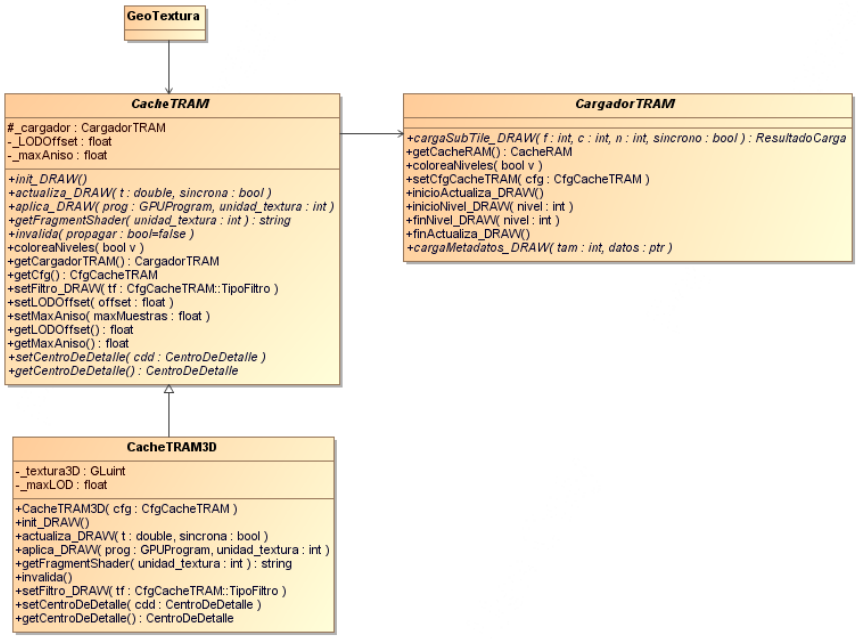


Figura 4.7: CacheTRAM3D. Diagrama UML de las clases CacheTRAM y relacionadas.

CacheTRAM construye la función GLSL `calculaFragment` que se utilizará desde el *fragment shader* para obtener el valor de la textura en cada posición. El código fuente en lenguaje GLSL se devolverá a través del método `getFragmentShader`. La clase CacheTRAM se encargará también de crear y asignar valor a los parámetros de tipo *uniform* que utilizará dicha función GLSL.

Tanto las funciones GLSL como los parámetros que utilizan se construyen en tiempo de ejecución a medida para un motor de textura concreto con su configuración específica. Por este motivo no se comparten entre diferentes

texturas, y para evitar conflictos en los identificadores, los de cada textura llevan el prefijo `tex`n`_` donde n es el número de la unidad de texturizado a la que está asociada. Así por ejemplo la función principal de la textura asociada a la primera unidad de texturizado se llamará `tex0_calculaFragment`.

Para mayor legilibidad, en el resto de este texto se hará referencia a los identificadores (ya sean nombres de funciones, parámetros, variables globales, macros del precompilador, etc.) omitiendo el prefijo, salvo en extractos del código GLSL generado por el sistema y mostrados como ejemplo, donde se utilizará habitualmente el prefijo `tex0_`.

La interfaz de la superclase CacheTRAM se detalla en el apéndice A.

En este trabajo se ha realizado el diseño y la implementación completos de una CacheTRAM, denominada **CacheTRAM3D** que será la que se describa en las siguientes secciones. Sin embargo, no es la única opción, podrían utilizarse otras cachés compatibles con la interfaz definida en la superclase.

Durante el desarrollo de este trabajo, se estudiaron otras alternativas al uso de texturas 3D, como la posibilidad de emplear las texturas cúbicas utilizadas habitualmente en OpenGL como mapas de reflexión del entorno. El objetivo era obtener un espacio de textura amplio en una única textura para albergar todo el *clipmap* en ella. Sin embargo, aunque las texturas cúbicas suelen permitir un tamaño máximo de textura contigua mayor que las texturas 3D, la superficie total disponible era inferior.

Al utilizar el direccionamiento toroidal, descrito en la sección 2.6.3, estableciendo el parámetro `TEXTURE_WRAP_x` en modo `REPEAT`, y el filtrado bilineal por *hardware*, la solución más adecuada resultó ser el uso de texturas 3D de tamaño, en ancho y alto, igual al tamaño de ventana cacheada, tal y como se explica en la siguiente sección, donde se describe la estructura de la caché implementada finalmente en el prototipo.

Una de las limitaciones más importantes para el uso de texturas 3D tal y como se plantea era el tamaño máximo de textura disponible por algunos sistemas gráficos o su *software* controlador. Por ejemplo, las tarjetas Nvidia GeForce hasta la serie 7 tenían una límite al tamaño de texturas 3D no superior a 512 *texels* en cada eje. En la siguiente generación este tamaño se amplió a 4096 *texels* de lado, un tamaño más que suficiente para la mayoría de aplicaciones del sistema propuesto. De hecho, en caso de utilizar un tamaño de ventana de ese orden, la limitación más importante sería el ancho de banda necesario para mantener la caché actualizada.

Una alternativa al uso de texturas 3D sería el uso de *arrays* de texturas [121], una característica que se hizo disponible durante el desarrollo de este trabajo, con la publicación de la especificación Direct3D 10 [42] de Microsoft y coincidiendo también con la generación de tarjetas gráficas GeForce serie 8 de Nvidia. Una de las aplicaciones de demostración de Nvidia para esta familia de *hardware* gráfico propone el uso de *arrays* de texturas para una implementación sencilla de *clipmaps* [118]. El motor de texturas virtuales dinámicas realizado en este trabajo podría realizarse de forma equivalente

mediante *arrays* de texturas en lugar de texturas 3D, y es un trabajo que se ha planteado para el futuro con el fin de analizar las diferencias de rendimiento que puedan existir entre ambas implementaciones, que posiblemente sería dependiente del *hardware* gráfico y de la versión de los controladores utilizados. En cualquier caso no supone ninguna diferencia en el diseño planteado, por lo que el resto del sistema permanecería inalterado.

A continuación se detallará el diseño de la CacheTRAM utilizada para las pruebas: CacheTRAM3D. Se comienza describiendo la estructura de la caché y posteriormente se analizarán por separado las dos tareas fundamentales: actualización y *render*.

4.3.3. Estructura de la caché

La estructura de la caché, basada en el concepto de *clipmap*[144] ya descrito en la sección 2.6.3, estará compuesta por dos partes diferenciadas: la **pirámide** del *clipmap* y la **pila** del *clipmap*. La primera se caracteriza por contener todos los niveles que estarán cargados enteramente en TRAM, mientras que la segunda, que corresponderá a los niveles de mayor detalle, se compondrá del subconjunto del espacio virtual completo de la textura que estará almacenado en la caché para cada nivel. Denominaremos **ventana cacheada** de cada nivel al área rectangular que ocupa este subconjunto del nivel completo.

Para almacenar toda la caché en una única textura, de forma que no se utilice más que una unidad de texturizado, y sin limitaciones importantes de tamaño, se ha utilizado una textura 3D donde el ancho y alto corresponden al tamaño de ventana utilizado. Además, el uso de una única unidad de texturizado minimiza el número de asociaciones de texturas (`glBind`), que se reducen a una, evitando por lo tanto la necesidad de agrupar la geometría según la textura que corresponda aplicar a cada zona.

Todos los niveles de la pila corresponderán a una capa de la textura 3D, de forma que la profundidad de dicha textura 3D será igual al número de niveles de la pila más dos para la pirámide. De estos dos últimos, uno contendrá el nivel de mayor detalle de la pirámide y el otro el resto de los niveles. Esta última capa contendrá necesariamente al menos la mitad libre, de forma que se podría aprovechar ese espacio para almacenar información disponible para el *shader* del motor de texturizado[4]. En la figura 4.8 se ilustra la estructura de la caché respecto a la pirámide de la textura virtual completa.

La cantidad máxima de memoria necesaria para la caché de una textura virtual de $m \times n$ *texels* se puede calcular mediante la ecuación 4.1, donde el tamaño de ventana es $v_x \times v_y$ *texels*.

[4]De hecho, algunas de las implementaciones realizadas durante este trabajo, utilizaban ese espacio para almacenar los metadatos sobre las zonas válidas de la caché. Estas implementaciones fueron descartadas posteriormente en favor de la implementación final descrita en este documento.

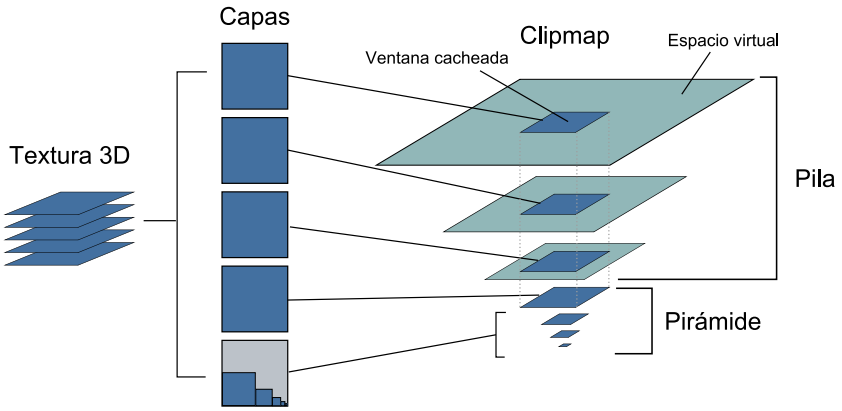


Figura 4.8: CacheTRAM3D. Distribución de la caché con estructura de *clipmap* en las capas de una textura 3D.

$$mem = (\log_2(\text{máx}(m,n)) - \log_2(\text{máx}(v_x,v_y)) + 2)v_x v_y \qquad (4.1)$$

La ventaja de esta estructura es que almacena un subconjunto de la textura virtual completa, suficiente para obtener el máximo detalle en la visualización, con tan sólo una mínima fracción de la memoria que haría falta para albergar dicha textura completa. La cantidad de memoria necesaria viene dada por la resolución de la pantalla, que limita el número de pixels que se pueden visualizar. Si se observa un detalle muy alto será una región geográfica pequeña, mientras que si se observa una región amplia el detalle será más reducido.

Con un tamaño de ventana adecuado y una correcta ubicación del centro de detalle, el resultado visual será indistinguible del que obtendríamos disponiendo de la textura completa cargada en TRAM.

El tamaño de ventana se debe escoger de forma que supere la resolución de la ventana de *render*. Habitualmente utilizamos un tamaño 2048×2048 *texels*, lo cual proporciona un margen de seguridad más que suficiente para las resoluciones de pantalla más habituales. Para la evaluación del sistema desarrollado se ha utilizado una resolución de pantalla de 1280x1024 pixels.

Según se aumenta la resolución de la textura, su tamaño aumenta exponencialmente, mientras que el tamaño de la caché necesaria para albergar la zona útil de dicha textura crece linealmente (ecuación 4.1).

Así si consideramos, por ejemplo, una textura de todo el planeta con una resolución uniforme de aproximadamente 0,0097 segundos de arco que corresponde, tras escalarlo a la potencia de dos más cercana, a una imagen de $134217728 \times 67108864$ *texels* ($2^{27} \times 2^{26}$), necesitaríamos un espacio de almacenamiento de 10,7 *petatexels* incluyendo la pirámide de *mipmap* completa. Este enorme espacio virtual se puede cachear en 72 *megatexels*, asumiendo un tamaño de ventana de 2048×2048, lo cual encajaría perfectamente en

las tarjetas gráficas domésticas actuales.

En comparación con la implementación de Tanner [144], basada en *hardware* específico de SGI, se necesita una cantidad de memoria ligeramente superior. Sin embargo, esta diferencia resulta irrelevante dadas las capacidades de memoria disponibles en el *hardware* gráfico actual. Por la naturaleza de la textura 3D donde se almacena la caché, el espacio adicional no es más que el producido por el efecto de fragmentación de memoria, puesto que forzosamente se debe reservar un número entero de capas en dicha textura. Ese espacio sobrante es por tanto inferior al tamaño de ventana ($v_x \times v_y$ *texels*). En cualquier caso, el uso de memoria es notablemente más reducido que el de otras aproximaciones como la de Ephanov [68], que utiliza una técnica de doble *buffer* o la de Seoane [137], que utiliza *mipmaps* propios para cada nivel de la pila.

Para facilitar las operaciones con la caché, se divide el espacio virtual de la textura en una colección de matrices (una por cada nivel de la pirámide) de fragmentos cuadrados que denominaremos **teselas** (ver figura 4.9). Una tesela será el "cuanto" de memoria a efectos de transferencia entre la caché en RAM y la caché en TRAM.

Los niveles de la pila almacenan una ventana rectangular del espacio virtual completo de la textura en ese nivel. Dado que esta ventana cacheada se va desplazando por el espacio virtual de forma continua, se minimiza el número de transferencias de memoria y se maximiza su reutilización mediante el uso de direccionamiento toroidal, tal y como ya se explicó en la sección 2.6.3.

Todas las ventanas de la pila son de igual tamaño en número de *texels*[5]. La posición de la ventana cacheada dentro del espacio virtual de cada nivel de detalle se almacena mediante un parámetro que hemos denominado **desplazamiento global**. Debido al direccionamiento toroidal, el espacio contiguo reservado para un nivel de la caché no siempre corresponderá con la ventana en el espacio virtual, sino que tendrá un cierto desplazamiento, que hemos denominado **desplazamiento local**. El desplazamiento local se puede calcular, para ambos ejes, mediante la ecuación 4.2

$$\begin{aligned} dl_x &= dg_x \mod v_x \\ dl_y &= dg_y \mod v_y \end{aligned} \tag{4.2}$$

donde dl es el desplazamiento local, dg el desplazamiento global y v el tamaño de la ventana, indicando todos estos valores en número de teselas.

Aunque técnicamente no hay ningún problema en manejar ventanas rectangulares, lo habitual es que sean cuadradas. El objetivo de la ventana es cachear una zona alrededor de un centro de detalle. Ya que esta ventana es un rectángulo en espacio textura, no podemos conocer a priori la orientación

[5]Cuando se haga referencia al "tamaño de ventana" en lo sucesivo a lo largo de este texto, se asume que será el número de *texels* de lado de una ventana cuadrada.

que va a tener en pantalla. Por lo tanto, para cubrir lo mejor posible la zona visible en cualquier situación de la cámara, se utilizan ventanas cuadradas.

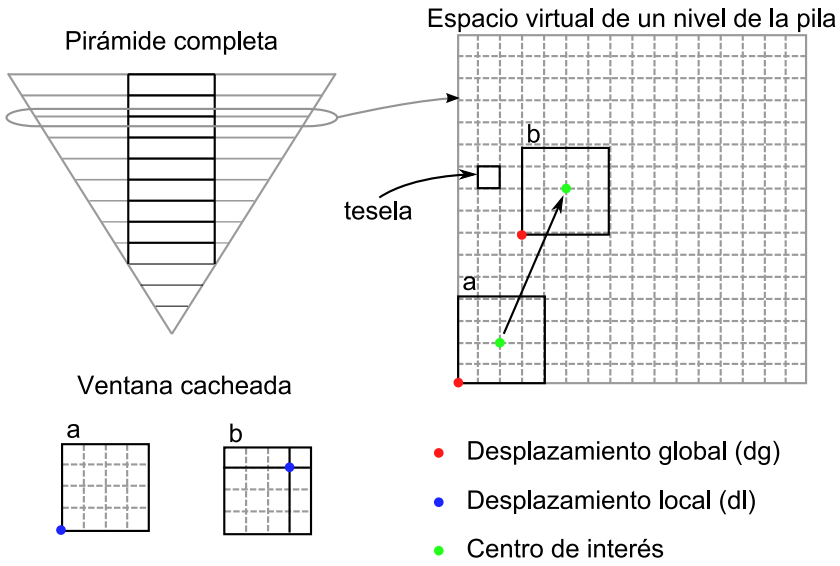


Figura 4.9: División del espacio virtual en teselas, ventana cacheada, direccionamiento toroidal, y desplazamientos local y global.

Además de los datos propios del contenido de la textura, se almacenan una serie de metadatos que incluyen el desplazamiento global de cada nivel de la pirámide y otra información relativa al estado de cada una de las teselas presentes en la memoria de la caché.

El estado de una tesela contendrá la información de **validez** para indicar si esa zona de memoria contiene la información actualizada que corresponde a la zona correcta de la ventana cacheada. La tesela no será válida cuando todavía no se haya cargado desde el siguiente nivel de caché (CacheRAM) o cuando no esté disponible en el origen. Todos los metadatos asociados a las teselas cacheadas se describen en la sección de actualización (4.4).

4.3.4. *Render*

El proceso de *render* es el fin último de la caché en TRAM. La caché contendrá los datos actualizados de la zona de interés, y éstos serán utilizados para texturizar la geometría dibujada. Entendemos por proceso de *render* el que comprende las siguientes operaciones:

1. Almacenar el estado del sistema gráfico para su posterior restauración (`guardarEstado_DRAW`).

2. Activar los *shaders* correspondientes a la textura, efecto o combinación de ellos, así como establecer los parámetros que esos *shaders* necesiten

para su correcto funcionamiento, activar unidades de texturizado y asociar (`glBind`) las texturas pertinentes (`aplica_DRAW`).

3. Enviar las órdenes de dibujado de primitivas geométricas al sistema gráfico. Esta tarea corresponde al motor de geometría del terreno, por lo que queda fuera del dominio de trabajo del sistema descrito y no se le impone ninguna dependencia ni restricción, ambos sistemas están completamente desacoplados.

4. Restaurar el estado del sistema gráfico (`restaurarEstado_DRAW`).

Este proceso se realiza fundamentalmente en la GPU, mediante un *shader* o programa por *fragment*. Para posibilitar el uso de la textura virtual desde cualquier otro *shader* más complejo o su combinación con otras texturas (no necesariamente virtuales), todas las operaciones se han encapsulado en una función escrita en lenguaje GLSL cuyo código fuente se genera, compila y enlaza en tiempo de ejecución, ya sea durante la fase de inicialización del motor de textura o en una reconfiguración posterior.

Esta función, denominada `calculaFragment`, se encarga de realizar las siguientes tareas:

- Calcular el nivel o los niveles de detalle de textura a los que se necesita acceder.
- Comprobar en la caché la disponibilidad de los *texels* en los niveles de detalle necesarios y en su caso calcular el nivel disponible más adecuado.
- Realizar los accesos necesarios a los *texels* adecuados de la textura 3D (direccionamiento de la caché).
- Combinar los *texels* obtenidos de la textura 3D para realizar el filtrado que corresponda.

Los detalles de diseño e implementación de estas tareas dentro del motor de *render* que se ha implementado como banco de pruebas para las técnicas desarrolladas en esta tesis doctoral se detallan en el apéndice B.

En dicho apéndice se detalla la estructura del *shader* utilizado para el *render*, los detalles de filtrado de texturas, tanto isotrópico (proximidad, bilineal, trilineal) como anisotrópico, y finalmente se establece una comparación entre los tipos de filtrado soportados en esta técnica y los definidos en las especificaciones de OpenGL[135].

4.4. Actualización síncrona

El proceso de actualización tiene por objetivo mantener las cachés cargadas con la información necesaria según la situación del centro de detalle.

Este proceso supone varias decisiones que afectarán a la eficacia de las cachés y por tanto a la calidad final de la visualización, aunque nunca deberían afectar al rendimiento interactivo del sistema, puesto que los procesos más lentos se realizarán de forma asíncrona.

Comenzando con el primer nivel de caché (`CacheTRAM`), se debe determinar qué niveles se actualizan y en qué orden. Dentro de un nivel de detalle, la forma de realizar las cargas puede seguir dos filosofías principalmente:

- Cargar regiones de textura de tamaño variable, invalidadas por desplazamiento del centro de detalle, tal y como se describe en el trabajo de Tanner *et al.*[144].

- Discretizar el espacio virtual de la textura completa en bloques de tamaño fijo (generalmente cuadrados), tal y como se describe en el trabajo de Seoane *et al.*[137].

La primera aproximación puede mejorar el rendimiento en determinadas circunstancias, pero a costa de aumentar la complejidad del proceso de actualización y dificultar el control del tiempo de carga, lo cual en la mayoría de los casos termina por afectar negativamente al rendimiento.

La solución que se ha tomado como base es la segunda de las descritas, tal y como ya se ha visto cuando se describió la división del espacio virtual en teselas en el apartado de estructura de la caché. Esta aproximación garantiza que el tamaño de bloque transferido maximice la velocidad de transferencia, evitando transferencias ineficientes y dejando por lo tanto más tiempo disponible para otras tareas.

En algunos casos, se puede utilizar una aproximación híbrida, de manera que se carguen simultáneamente varios bloques contiguos formando una zona rectangular. Esto es especialmente interesante en el caso de *render* a textura a partir de datos vectoriales, puesto que a diferencia de la carga de datos *raster*, no se puede asumir un tiempo de carga fijo.

El proceso de actualización se puede dividir en dos partes: una que se realiza de forma **síncrona** con el proceso de *render*, y por tanto su rendimiento se considera crítico y se limita en tiempo; y una segunda parte **asíncrona**, encargada de realizar las cargas de disco, que se realiza en segundo plano por tratarse de una tarea excesivamente lenta que afectaría al rendimiento interactivo.

En cada iteración de la parte síncrona del proceso de actualización (método `actualiza_DRAW`), se desempeñan las siguientes tareas, que la clase `GeoTextura` delega completamente en `CacheTRAM`:

1. Invalidar las teselas que hayan caducado. El sistema de gestión de caducidades será descrito en detalle más adelante en esta sección.

2. Invalidar las teselas de la caché que han quedado fuera de la ventana debido al desplazamiento del centro de detalle desde el último fotograma. Los bloques de memoria de las teselas que quedan fuera de la ventana, debido al direccionamiento toroidal utilizado, se reutilizan para las nuevas teselas cambiando su ubicación dentro del espacio virtual (ver figura 2.32).

3. Activar (en OpenGL) la textura que contiene la caché para poder actualizarla.

4. Notificar al cargador que comienza un ciclo de actualización, para que realice las operaciones que estime oportunas (por ejemplo, activar un FBO, tal y como se describe en el apendice E)

5. Determinar qué niveles van a ser actualizados y en qué orden.

6. Para cada nivel de detalle, en el orden determinado (por ejemplo, de menor a mayor detalle), y mientras no se haya superado el tiempo asignado para la actualización y no esté completamente actualizado:

   a) Notificar al cargador que comienza la actualización de ese nivel en concreto, para que realize las operaciones necesarias (por ejemplo, asociar al FBO la textura correspondiente a ese nivel).

   b) Generar la lista ordenada de teselas a cargar, según el estado actual de la caché, la posición del centro de detalle, la dirección y velocidad de la cámara del *render* y cualquier otro parámetro que pueda afectar. El algoritmo para determinar el orden en que se cargarán las teselas se describe más adelante.

   c) Para cada tesela de la lista ordenada, mientras no se haya superado el tiempo asignado a la actualización:

      1) Actualizar la tesela. El éxito de este proceso no está garantizado, y dependerá de la disponibilidad de los datos en el sigiuente nivel de caché (`CacheRAM`) o incluso en el origen de datos. Este proceso se describe en detalle más adelante.

   d) Notificar al cargador que ha finalizado la actualización del nivel, para que realice las operaciones necesarias.

   e) Actualizar los metadatos del nivel. La gestión de los metadatos de `CacheTRAM` se describe en detalle más adelante.

7. Notificar al cargador que ha terminado el ciclo de actualización, para que realice las operaciones necesarias.

8. Actualizar la información de los límites válidos en cada nivel de detalle, que definen el área rectangular usable en el *render*.

9. Actualizar otros parámetros que informan al *render* acerca del estado de la caché, como el nivel de detalle máximo disponible con información útil (`maxLOD`) que evita cálculos innecesarios al *fragment shader*, mejorando el rendimiento del sistema.

A continuación se describen los principales aspectos de diseño y las decisiones que se han tomado para optimizar el resultado y cumplir los objetivos planteados en el capítulo anterior.

4.4.1. Orden de carga de los niveles

La primera decisión importante consiste en decidir qué niveles se van a cargar y en qué orden.

La aproximación clásica a este problema ha sido la carga de abajo hacia arriba (figura 4.10a), comenzando por los niveles con menor detalle, cuya ventana cubre una extensión geográfica mayor, para ir refinando el detalle progresivamente en zonas geográficas cada vez más reducidas, en torno al centro de detalle. Este es el comportamiento que siguen la gran mayoría de los sistemas existentes, como los descritos por Cosman [55] y Tanner [144].

Se trata de una solución excelente para información estática, puesto que se dispone rápidamente de información en cualquier parte del terreno y se refina siguiendo los movimientos del centro de detalle de forma que siempre se tiene información relativamente adecuada para el *render*. En los casos de movimientos rápidos, que desborden la capacidad de transferencia del sistema, se reducirá momentáneamente y moderadamente la calidad de la textura, que se recuperará de forma progresiva.

Sin embargo, en un sistema de texturizado dinámico como el desarrollado en este trabajo no resulta adecuado porque puede dar lugar a situaciones en que la cámara esté situada en una posición cercana al terreno y enfocada en un detalle fino que nunca se alcanzará porque el tiempo de actualización se está empleando en recargar continuamente los niveles de detalle inferiores que no son necesarios en absoluto para esta vista. Este problema se hace patente cuando la frecuencia de actualización de los datos y el nivel de detalle máximo son elevados.

Cuando se maneja información dinámica con un ciclo de actualización reducido, es importante determinar cuáles son los niveles que se necesitarán en el *render* y comenzar la carga por dichos niveles y no por los más bajos. No obstante, este cálculo, aunque evidente dentro de un *fragment shader* donde se calcula el nivel de detalle que se solicitará a la textura, es necesario hacerlo en la CPU durante la fase de actualización, lo cual lo dificulta seriamente y lo hace bastante costoso en tiempo. Por este motivo, y para evitar deteriorar el rendimiento del sistema, se realizan aproximaciones de las necesidades del *render*, primando la economía de cálculo sobre la precisión de los resultados.

Para definir el orden de carga de los niveles, se comienza por determinar

el nivel de máxima prioridad según la vista actual y se realizan las cargas a partir de ese nivel, alternando los niveles inferiores con los superiores (ver figura 4.10b). De esta forma, cuando el proceso de actualización de la caché se satura y no se pueden mantener todos los niveles actualizados, se mantendrán los que tienen mayor importancia en la vista actual.

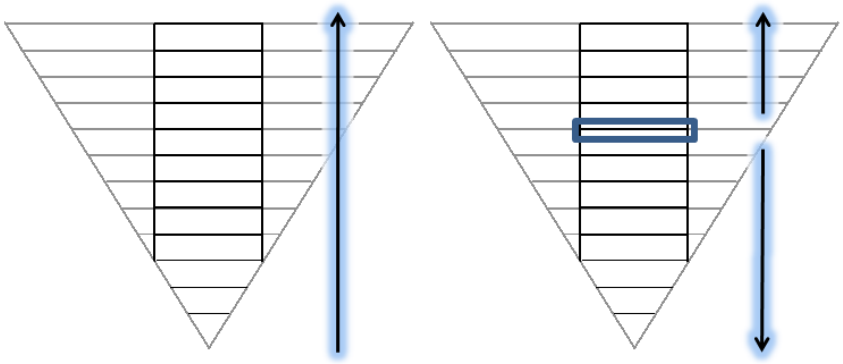


Figura 4.10: Orden de carga de niveles. a) Aproximación clásica. b) Aproximación propuesta para datos dinámicos.

Este cálculo está en relación directa con la ubicación de la pila flotante, descrita en la sección 4.7. En el apéndice C se describe un algoritmo para el cálculo del nivel de detalle principal según la vista del usuario.

4.4.2. Orden de carga de las teselas dentro de un nivel

A la hora de definir el orden de carga de las teselas dentro de un nivel de detalle de la textura se han tenido en cuenta los siguientes objetivos:

- Cargar primero las zonas más cercanas al centro de detalle, que será muy probablemente el que ocupe la zona de pantalla que tiene la mayor atención por parte del usuario.
- Cargar progresivamente zonas rectangulares completas en torno al centro de detalle (formar tan rápido como se pueda zonas usables directamente en el *render* lo más amplias posible).
- Dar mayor prioridad a aquellas zonas que, de mantenerse la dirección de movimiento actual, vayan a resultar útiles por más tiempo.

El sentido de cargar zonas completas rectangulares es que, para mantener la simplicidad en el *fragment shader*, la información de disponibilidad de teselas válidas en caché para un nivel se le proporciona en forma de los límites de una zona rectangular. El motor de *render* asumirá que el rectángulo delimitado tiene información válida y actualizada, lista para texturizar. Esto

hace muy eficiente durante el *render* la operación de comprobar la disponibilidad de textura en caché para unas coordenadas de textura y un nivel de detalle determinados.

En cada ciclo de actualización y para cada nivel se construye una lista ordenada de teselas a cargar. La importancia de este orden de carga radica en que es posible, puesto que se prioriza el rendimiento interactivo frente al resto de características del sistema, que no se disponga de tiempo suficiente en un ciclo para actualizar completamente la caché, de forma que algunas teselas quedarán invalidadas o no disponibles.

Además, como el movimiento de la cámara es contínuo, en cada ciclo el centro de detalle puede estar situado en una posición distinta. Sin embargo, se asume una cierta continuidad en la ubicación de este centro de detalle, que en un alto porcentaje de los casos sigue una trayectoria sin cambios excesivamente bruscos y en casi ningún caso hay discontinuidades en su movimiento. En caso de haber discontinuidades, evidentemente, dependiendo de la distancia del desplazamiento, se invalidará y por tanto habrá que recargar una parte mayor de la caché y muy posiblemente habrá pérdidas de detalle apreciables, detalle que será recuperado progresivamente en un tiempo breve.

La vida útil de una tesela puede ser muy reducida en casos de alta velocidad en el movimiento de la cámara, por lo que si no se es muy escrupuloso a la hora de decidir el orden de carga, se pueden dedicar preciosos recursos a la carga de teselas que nunca llegarán a usarse. Hay que decir también que es precisamente en estos movimientos veloces donde el espectador apreciará menos las pérdidas de detalle y en todo caso resultarán algo natural, un efecto similar al desenfoque por movimiento de una cámara en la realidad.

El algoritmo utilizado en este trabajo para determinar el orden de carga de teselas, ilustrado en la figura 4.11, se basa en la carga progresiva del borde de cuadrados concentricos (asumiendo, como ya se ha detalado anteriormente, que la ventana cacheada es cuadrada en *texels*), que por analogía denominaremos **anillos**.

Tomando el orden desde dentro (zona más cercana al centro de detalle) hacia afuera, cada anillo se divide en cuatro brazos, diferenciados según la dirección de avance de la cámara. Denominaremos estos brazos como **frontal**, **fronto-lateral**, **lateral-trasero** y **trasero** (figura 4.12).

El brazo frontal es el que se encuentra en la dirección principal de avance, y es el primero que se cargará, comenzando por la tesela del extremo hacia el que avanza la cámara. Esta tesela no es la de la esquina de avance (la que posee una mayor esperanza de vida útil en el anillo que se está procesando) sino que es la inmediatamente anterior. Tras ella se cargarán el resto de teselas de ese borde del anillo, a excepción de la última, que corresponde al brazo lateral-trasero.

El motivo de cargar las teselas de esta forma es ampliar la zona útil completando rectángulos en torno al centro de detalle y dando prioridad a

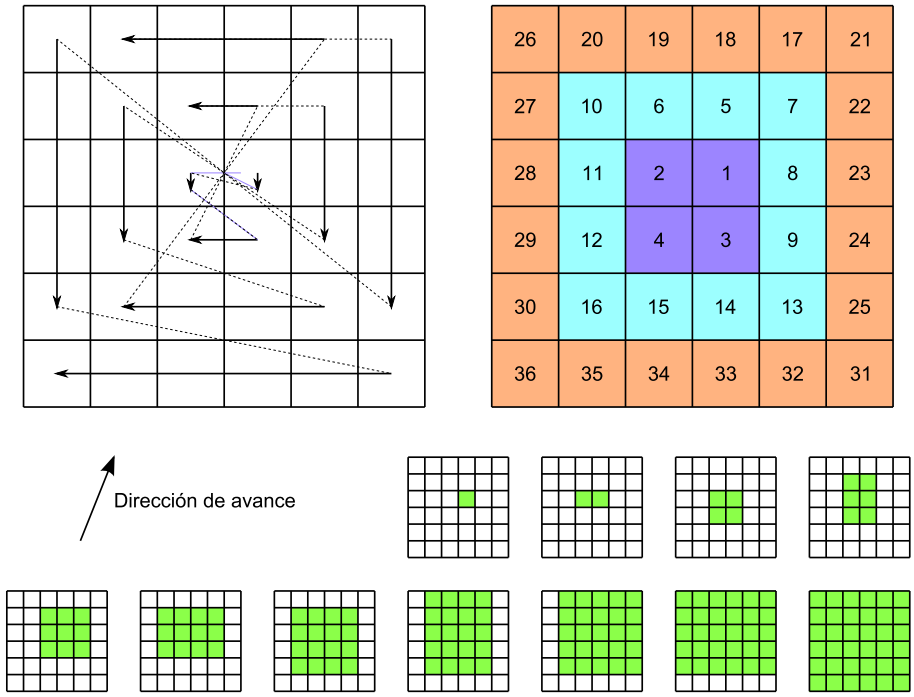


Figura 4.11: Orden de carga de las teselas dentro de un nivel de detalle. En este jemplo se muestra la carga de los tres primeros anillos concéntricos de teselas para un nivel determinado asumiendo una posible dirección de avance.

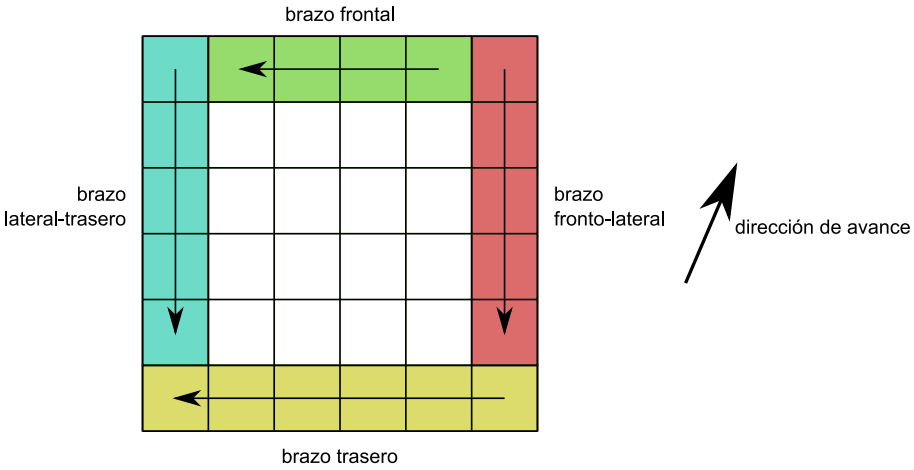


Figura 4.12: División de un anillo concéntrico en cuatro brazos y orden de carga de sus teselas para una dirección de avance de ejemplo.

la dirección de avance. En la parte baja de la figura 4.11 se muestra la secuencia de rectángulos disponibles para el *render* en los tres primeros anillos concéntricos. En este caso se completan 11 rectángulos, puesto que el primer

anillo, al estar compuesto únicamente por cuatro teselas, generará sólo tres rectángulos (el brazo frontal no contiene ninguna tesela).

El segundo brazo (fronto-lateral), corresponderá con la dirección de avance secundaria, y carga todas las teselas de ese lateral del anillo a excepción de la última (que corresponde al brazo trasero).

El tercer brazo (lateral-trasero), está en el borde opuesto al anterior, y cargará todas las teselas de ese lateral del anillo a excepción de la última (que corresponde al brazo trasero).

En resumen, este orden de carga persigue cumplir los objetivos planteados anteriormente: dar prioridad a las zonas cercanas al centro de detalle, maximizar la esperanza de vida cargando primero las teselas en la dirección de avance de la cámara y formar rectángulos de zona actualizada en los niveles de la caché, de forma que puedan ser utilizados directamente en el *render*.

4.4.3. Gestión de los metadatos de la caché en TRAM

Para la correcta gestión de la caché en TRAM, es necesario almacenar una serie de metadatos acerca del estado de actualización de los diferentes niveles de detalle de textura y de las teselas dentro de cada uno de esos niveles de detalle.

Esta información se encapsula en dos clases que representan los conceptos de caché de un nivel de detalle de la textura y de las teselas que componen esa caché respectivamente (`EstadoTesela` y `MatrizTeselas`), tal y como se muestra en la figura 4.13.

Para cada nivel de detalle de la textura virtual se almacena en `MatrizTeselas` la siguiente información relativa a la caché de dicho nivel:

- Dimensiones del espacio virtual de la textura, medidas en número de teselas en ancho y alto.
- Dimensiones de la ventana cacheada en número de teselas en ancho y alto.
- Area geográfica cubierta por el espacio virtual.
- Desplazamiento global. Desplazamiento de la ventana cacheada, dentro del espacio virtual, en número de teselas.
- Desplazamiento local. Origen de la ventana dentro de la memoria (textura) reservada para cachear el nivel, en número de teselas.
- Indicador de si el nivel está completo (todas las teselas están actualizadas con información válida y no caducada).

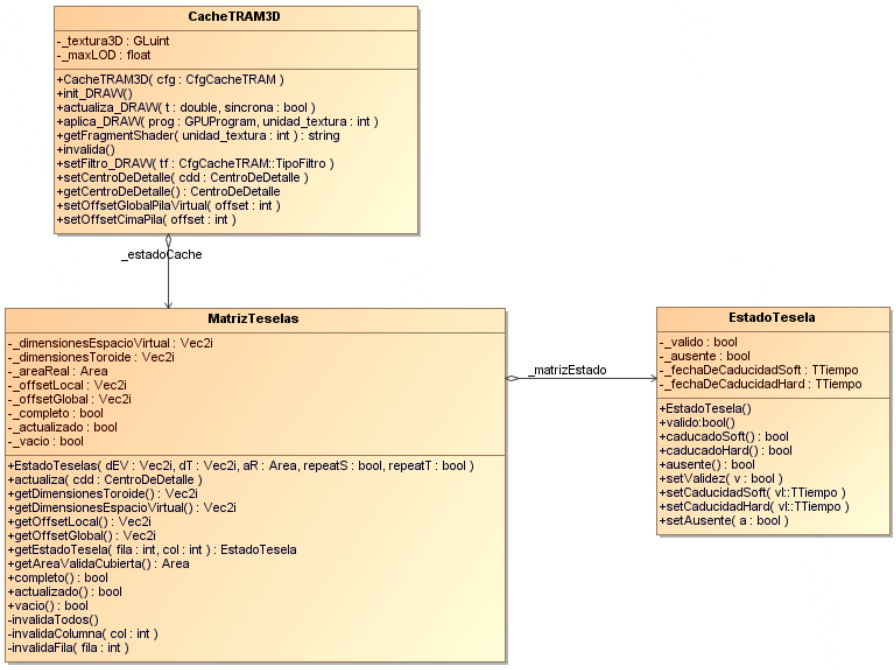


Figura 4.13: Diagrama UML de las clases que encapsulan el estado de la caché.

- Indicador de si el nivel está actualizado (no quedan cargas pendientes para el nivel). Un nivel actualizado no necesariamente tiene por qué estar completo. Puede haber partes de la ventana cacheada que no estén disponibles en origen.
- Indicador de si el nivel está vacío (no contiene ninguna tesela con datos útiles). Esto ayuda a descartarlo directamente en el *render*.

Además, dicha clase contiene la matriz de estados de cada tesela de la ventana cacheada. La información asociada a la tesela incluye las siguientes propiedades:

- Válido.
- Ausente.
- Tiempo límite para solicitar su recarga (`fechaCaducidadSoft`).
- Tiempo límite para considerarlo no usable (`fechaCaducidadHard`).

La validez de una tesela indica que en la zona de memoria de la caché que tiene asignada se ha cargado la información de la zona geográfica que le corresponde según la posición actual del centro de detalle. Esto no sucede en las teselas que salen fuera de la ventana cuando se desplaza el centro de

detalle, por lo que se invalidan inmediatamente. El hecho de que una tesela sea válida no quiere decir que sus datos sean usables, puesto que el tiempo de validez de la información cargada podría haber expirado, es decir, una tesela válida puede estar caducada.

La propiedad de ausencia de una tesela indica que esa zona de la textura no está disponible en el origen de datos. El almacenar esta propiedad evita la repetición continua de peticiones a la caché de segundo nivel (`CacheRAM`) para aquellos elementos que nunca estarán disponibles en dicha caché.

Sin embargo, esta condición no es eterna, las teselas marcadas como ausentes también tienen un tiempo de validez, de forma que cuando expire ese plazo se volverán a solicitar. Esto permite utilizar texturas virtuales con nivel de detalle heterogéneo y variable en el tiempo. Otra posible utilidad es poder aplicar texturas cuya generación todavía no se ha completado y disponer en el futuro de esa información que todavía no existe sin necesidad de forzar una recarga de la textura completa.

Por lo tanto, es importante indicar que para una mayor versatilidad se consideran de forma completamente independiente las condiciones de validez, ausencia y caducidad, de forma que ninguna de ellas implica un estado concreto en las otras, salvo que una tesela ausente nunca puede ser válida.

4.4.4. Actualización temporal

Además de la actualización espacial que se ha descrito hasta ahora, cuando se trabaja con datos dinámicos, se debe tener en cuenta la actualización temporal de los mismos.

Para solucionar este problema se ha creado un mecanismo de actualización basado en un tiempo de vida asociado a las teselas. Para una mayor flexibilidad, cada tesela tiene su propio tiempo de vida, independientemente del que tengan todas las demás. Este tiempo de vida viene indicado en el proceso de carga de la tesela desde el siguiente nivel de caché (que a su vez obtendrá dicho tiempo del origen de datos).

La información relativa al tiempo de vida de las teselas se almacena como un sello de tiempo que corresponde con la fecha de caducidad de la tesela y no como la duración o período de utilidad de esos datos. De esta forma se reducen los cálculos a realizar en la continua tarea de comprobar si se ha superado el período de utilidad de los datos.

Cuando la fecha de caducidad de una tesela se alcanza, el proceso de actualización determina que esa información está obsoleta y solicita su recarga al siguiente nivel de caché.

Además, para evitar caídas en el nivel de detalle visualizado cuando se alcanza la fecha de caducidad de las teselas, se ha diseñado un mecanismo de tiempos de caducidad en dos niveles. Con la caducidad simple asociada a las teselas, al alcanzar ese momento de caducidad, los datos se consideran obsoletos y se debe solicitar su recarga inmediatamente. De esta forma, cada

vez que se alcance el tiempo de caducidad será muy habitual que se pierda detalle visual momentáneamente, lo cual producirá unos parpadeos tan notables como molestos. Esto es debido a que en el tiempo del fotograma destinado a la actualización síncrona de la caché no se puede garantizar que haya tiempo suficiente como para actualizar todas las teselas caducadas en ese fotograma.

Este efecto se soluciona con el esquema de dos niveles, que funciona de la siguiente manera: una vez alcanzado el tiempo de caducidad del primer nivel, se solicita la recarga de los datos, pero se permite que se sigan utilizando mientras no se hayan recargado. El segundo nivel de caducidad es el que notifica que los datos ya no son aceptables y se deben dejar de utilizar en el *render*. Se garantiza por tanto un período de gracia para la carga de las teselas caducadas, que de esta manera se puede repartir entre varios fotogramas sin que se produzcan artefactos indeseados en la visualización.

Además de este mecanismo de actualización temporal que podemos denominar síncrono, existe la posibilidad de forzar de manera asíncrona la recarga de información, ya sea a nivel de tesela, de nivel de detalle o de la caché completa. Esto se realiza mediante llamadas a los métodos de invalidación de las clases correspondientes.

4.4.5. Continuidad en los bordes de la textura virtual

A la hora de gestionar la actualización de las cachés, es necesario tener en cuenta el comportamiento del espacio de la textura en los bordes. Tal y como se explica en los apartados sobre el *render* (sección 4.3.4 y apéndice B), en algunos casos existe continuidad en determinados bordes de la textura, de forma que se trata de un espacio esférico o toroidal en lugar de un espacio aislado.

La configuración asociada a una textura incluye los parámetros que indican la condición de repetición en cada eje de la textura (s y t). Estos parámetros se deben tener en cuenta a la hora de ubicar la ventana del espacio virtual cacheada en un nivel de la pila.

En la figura 4.14 se muestran varios ejemplos de la ubicación de la ventana cacheada según las condiciones de continuidad en los bordes de la textura virtual.

En el primer ejemplo no existe continuidad en ninguno de los dos ejes, por lo que cuando el centro de detalle se aproxima a los bordes, la ventana se mantiene ajustada a dichos bordes. Esta sería la situación en el caso de una textura de una zona de terreno aislada.

En el segundo ejemplo existe continuidad en el eje s, por lo que la ventana cacheada se distribuye entre los extremos izquierdo y derecho del espacio virtual, centrada en la posición del centro de detalle respecto al eje s pero no al eje t, donde la ventana se ajusta al borde superior de la textura. Esta sería la situación en el caso de una textura planetaria en coordenadas geodésicas

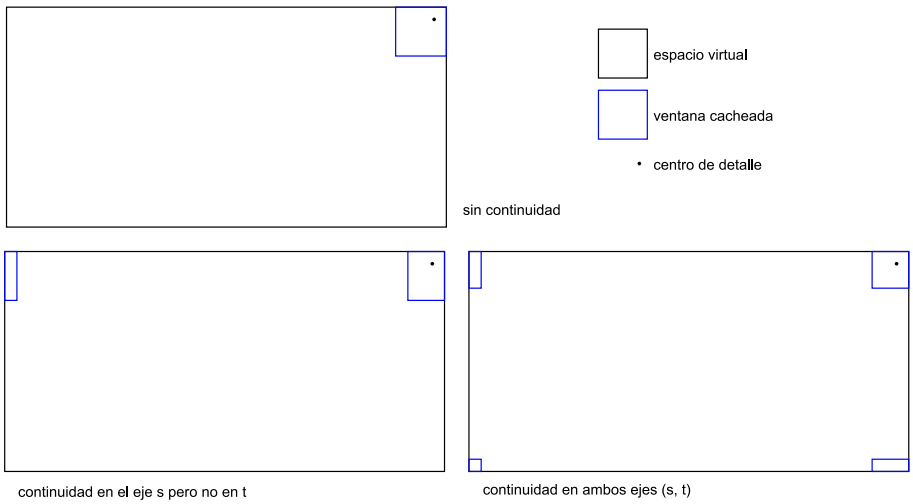


Figura 4.14: Ejemplos de ubicación de la ventana cacheada según las condiciones de continuidad en los bordes.

(longitud y latitud), donde hay continuidad este-oeste pero no norte-sur.

En el tercer ejemplo existe continuidad en ambos ejes, por lo que la ventana cacheada está centrada en torno al centro de detalle en ambos ejes y se distribuye por las cuatro esquinas del espacio virtual de la textura.

Por lo tanto, esta condición se debe tener en cuenta, además de en el proceso de *render* como ya se ha descrito, en las tareas de cálculo de los desplazamientos (local y global) de la ventana cacheada y en el direccionamiento de las matrices de estado de las teselas de cada nivel.

4.4.6. Proceso de actualización de una tesela

El proceso de actualización de una tesela se realizará por orden para cada una de las incluidas en la lista de actualización mientras no se haya consumido el tiempo asignado para la actualización síncrona, tal y como se describió al inicio de esta sección.

El orden de carga se decide según el algoritmo descrito anteriormente e ilustrado en la figura 4.11. Como las teselas a cargar se indican en el espacio local de la ventana cacheada, la primera tarea consiste en calcular la dirección de la tesela en el espacio virtual de la textura completa. Para esto es importante tener en cuenta las condiciones de continuidad recién descritas.

Una vez calculada la dirección correcta de la tesela en el espacio virtual del nivel de detalle correspondiente, se solicita al cargador (objeto de la clase `CargadorTRAM`, que se encargará de solicitarlo a la caché de segundo nivel (`CacheRAM`) y, en caso de conseguirlo, transferirlo a TRAM.

El cargador indicará el resultado de su solicitud, que será uno de los siguientes y tendrá las consecuencias descritas a continuación.

Cargado La tesela se ha cargado en TRAM con éxito. En este caso, se marca como válida y no ausente y se actualizan sus fechas de caducidad.

Pendiente Esta condición indica que la tesela no está disponible todavía en la caché de segundo nivel, bien porque todavía no se ha solicitado su carga o bien porque dicha carga está en curso pero aún no ha sido completada. En estos casos se deja el estado de la tesela intacto, de forma que se vuelva a solicitar en el próximo ciclo de actualización en caso de que todavía sea necesaria. El hecho de no cambiar el estado de la tesela es necesario para el correcto funcionamiento del mecanismo de doble caducidad, puesto que si la solicitud de carga se ha realizado debido a la caducidad de una tesela que por otra parte es válida, se puede seguir utilizando mientras se produce la recarga, a no ser que haya sido invalidada por superar el segundo tiempo de caducidad.

Error Se ha producido un error indeterminado en la carga de la tesela. Se marca el estado de dicha tesela como no válida y no ausente. Esta situación no se producirá en condiciones normales de funcionamiento, puesto que las teselas no disponibles en origen se consideran ausentes, como se indica a continuación.

Ausente La tesela solicitada no está disponible en el origen de datos. Se marca como ausente y no válida, y se actualizan sus fechas de caducidad para posibilitar una nueva solicitud en el futuro.

4.4.7. Tamaño de bloque para la actualización

Como ya se ha descrito, la actualización se realiza en bloques de tamaño fijo. En el primer nivel de caché estos bloques se han denominado teselas. El tamaño de tesela es crítico para el rendimiento del sistema y su correcta elección es por tanto de vital importancia.

El comportamiento habitual es que a mayor tamaño de bloque, mayor eficiencia en las transferencias. Sin embargo, los bloques muy grandes tienen la desventaja de que dificultan el control del tiempo de carga. Para una planificación de tiempos adecuada es necesario que el "cuanto" de carga (o de *render*, en caso de texturas procedimentales) no sea demasiado elevado, es decir, se necesita una granularidad suficientemente fina para un buen ajuste del tiempo de actualización en cada fotograma.

La elección del tamaño de tesela debe responder a un balance entre ambos aspectos: velocidad de transferencia y granularidad. Este parámetro es dependiente del *hardware* utilizado, por lo que sería recomendable realizar unos tests durante la instalación para determinar cuál es el tamaño óptimo para cada caso.

Las pruebas realizadas durante el desarrollo del sistema han indicado que existe un tamaño a partir de la cual no hay mejora en la velocidad de

transferencia o esta se puede considerar despreciable. Ese es el tamaño de bloque que se utiliza para las teselas, puesto que tamaños inferiores degradan notablemente la velocidad de transferencia y tamaños superiores, sin mejoras importantes en dicha velocidad, aumentan la granularidad y dificultan un ajuste fino del tiempo de carga.

Otro aspecto que afecta al rendimiento de las cargas es el formato de pixel utilizado para las texturas en la caché. En algunos casos, esto es evidente, puesto que los formatos de pixel pueden tener diferentes tamaños en número de bits por pixel. Sin embargo hay que tener en cuenta algunos detalles, como que el ancho del bus utilizado para las cargas implica que cargas de pixels de tamaño inferior al del bus pueden durar el mismo tiempo que las de pixels del tamaño del bus. Además, incluso dentro del mismo tamaño de pixel, formatos diferentes muestran diferentes rendimientos. Por ejemplo, utilizando cuatro canales, los tres de color (rojo, verde y azul) y un canal alfa para información adicional como opacidad o transparencia, muestra resultados distintos según el orden en que se organizen dichos canales. Así el formato de OpenGL `GL_BGRA` es más eficiente que el habitual `GL_RGBA`.

En las pruebas realizadas, el tamaño de bloque que se podría considerar óptimo para las cargas está en torno a los 128×128 *texels*, que es lo suficientemente pequeño para permitir un buen control del tiempo de actualización y está en el comienzo de la zona óptima de la curva de velocidad de transferencia.

En las gráficas 4.15 y 4.16 se muestran los tiempos de carga para diferentes formatos de pixel y diferentes tamaños de bloque en dos configuraciones de *hardware* (NVIDIA GeForce 7800GS y 8800GTS respectivamente).

En el caso de *render* a textura, el control del tiempo de "carga" no es tan fácil como en el caso de las cargas desde RAM, puesto que no se puede asumir un tiempo aproximadamente fijo para un tamaño de bloque determinado. Este tiempo es dependiente de la complejidad de los datos vectoriales en la zona que ocupa la tesela.

El tamaño de tesela también afecta al tiempo de *render* aunque de diferente forma. No existe ese umbral de tamaño a partir del cual no se consigue un aumento significativo en la velocidad de transferencia.

A mayor tamaño de tesela, mayor eficiencia en el *render*, lo cual junto con el hecho de que el tiempo es muy variable, hace muy útil el *render* de zonas de diferentes tamaños según la complejidad de la escena en cada momento. De hecho, el tamaño óptimo, si el tiempo de actualización lo permite, sería la ventana completa del nivel.

En el siguiente capítulo se analizará en profundidad el impacto del tamaño de tesela en las tareas de actualización, tanto para datos *raster* como vectoriales.

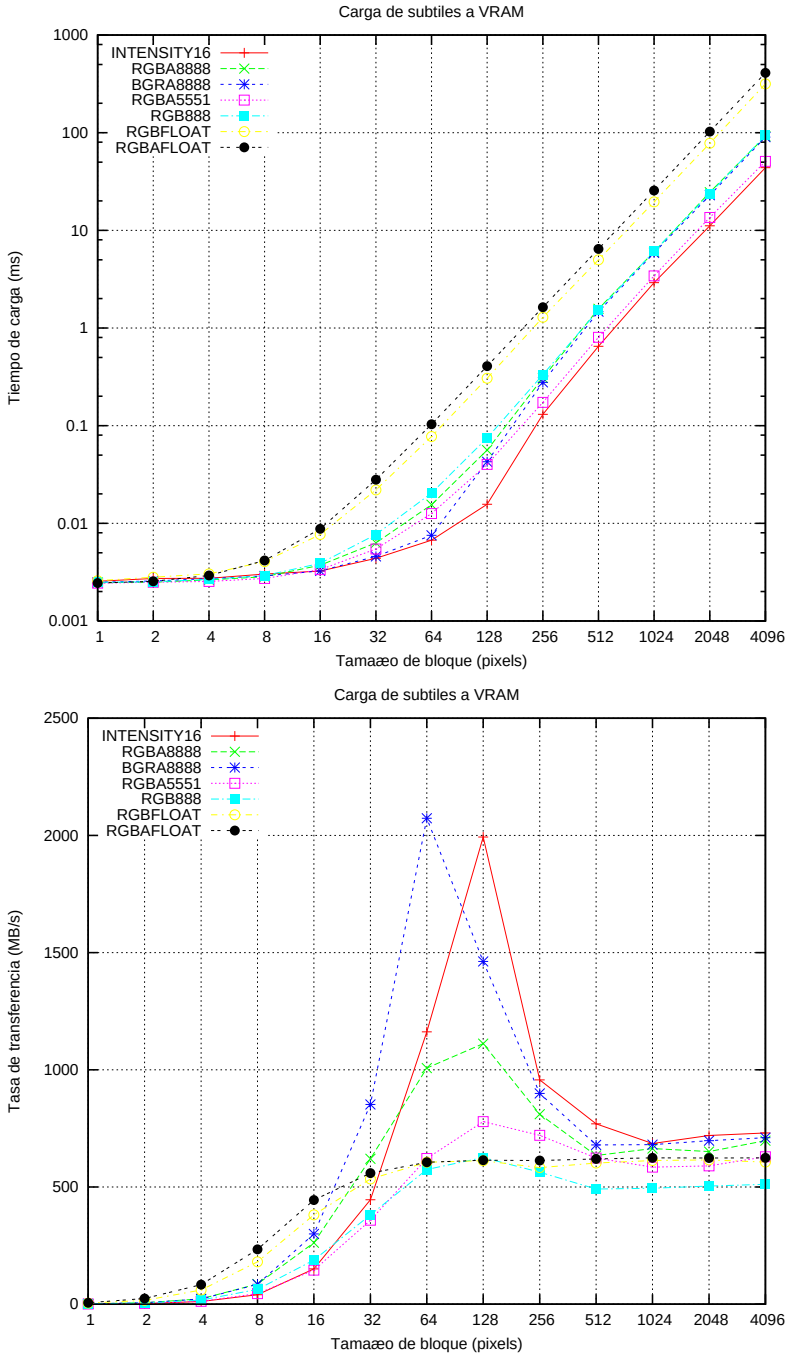


Figura 4.15: Tiempos de carga de bloques en TRAM y velocidades de transferencia. NVIDIA GeForce 7800 GS/AGP/SSE2 (OpenGL 2.1.0 NVIDIA 97.46 Linux x86).

4.5. Actualización asíncrona

La caché en TRAM carga las teselas que se necesitan de forma inmediata en el *render*. El origen de estas teselas puede ser la memoria principal (RAM)

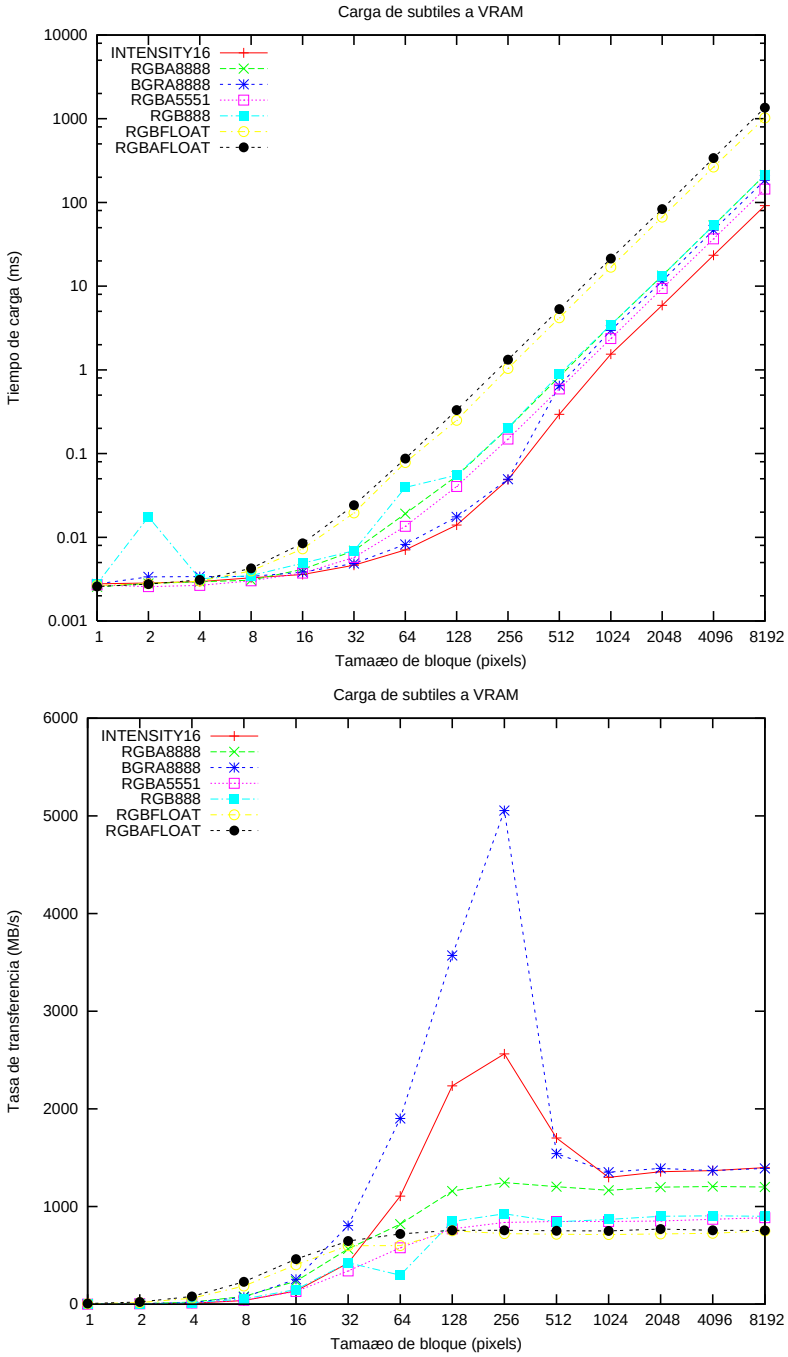


Figura 4.16: Tiempos de carga de bloques en TRAM y velocidades de transferencia. NVIDIA GeForce 8800 GTS/PCI/SSE2 (OpenGL 2.1.0 NVIDIA 97.55 Linux x86).

en el caso de información *raster* o la generación mediante *render* a textura en el caso de información vectorial. Este último caso se trata en detalle en la

sección 4.6, por lo que ahora nos centraremos en el caso específico de carga de datos *raster* desde RAM.

La velocidad de estas cargas viene determinada principalmente por la velocidad de transferencia del bus entre la CPU/RAM y la GPU/TRAM, actualmente PCIe 16x con una velocidad nominal de 8 GBytes/s. Para poder realizar estas cargas, la información necesaria debe estar disponible en la caché en RAM (CacheRAM).

La velocidad de carga del almacenamiento secundario a la RAM es varios órdenes de magnitud más lenta que entre RAM y TRAM, por lo que estas cargas se realizan en un proceso asíncrono que no paralice el hilo de ejecución principal, encargado del *render*.

Ya que se realizan las cargas en un proceso asíncrono y no se tienen por tanto unas limitaciones de tiempo tan estrictas, ni se necesita una granularidad tan fina en los tiempos de carga, los bloques cargados en la CacheRAM serán de tamaño superior a las teselas cargadas en TRAM. De esta forma se incrementa la tasa de transferencia, consiguiendo una mayor eficiencia en las cargas desde disco. Para diferenciar claramente las unidades de carga en una y otra caché, denominaremos ***buffers*** a los cargados en la CacheRAM, frente a las **teselas** cargadas en la CacheTRAM.

La carga de los datos bajo demanda en la CacheRAM, no es una buena opción, puesto que, aunque no paralizaría el *render*, sí se producirían retrasos importantes en la disponibilidad de los datos de textura para la zona y el nivel necesarios, con la consecuencia indeseable de pérdidas de detalle visual. Por este motivo, la CacheRAM se encarga de realizar una carga predictiva siguiendo el algoritmo descrito a continuación.

CacheRAM utiliza el tiempo disponible para cargar, en segundo plano, tanta información como pueda en torno al centro de detalle, aún cuando no se necesiten todavía los datos cargados. De esta forma, el usuario tendrá un margen amplio de movimiento con el detalle máximo disponible de forma inmediata o cuasi-inmediata. En resumen, el objetivo que se persigue es minimizar los fallos de caché.

De nuevo el orden de carga de los *buffers* (al igual que sucedía con la carga de teselas en TRAM) es crítico para obtener un rendimiento y una calidad óptimos. Dicho orden de carga se define mediante el algoritmo descrito a continuación e ilustrado en la figura 4.17.

Para cada nivel de detalle, se calcula el conjunto de *buffers* que forman una región rectangular cubriendo la ventana del terreno cacheada en TRAM para ese nivel. Este conjunto de *buffers* forman lo que denominaremos **zona A**.

El conjunto en forma de L de *buffers* que limitan con la zona A en la dirección de movimiento del centro de detalle será denominado **zona B**.

El resto de los *buffers* para ese nivel serán considerados como **zona C**.

Los *buffers* de la zona A son los necesarios inmediatamente para el *render*, por lo que deberían estar disponibles para su solicitud por el objeto de la clase

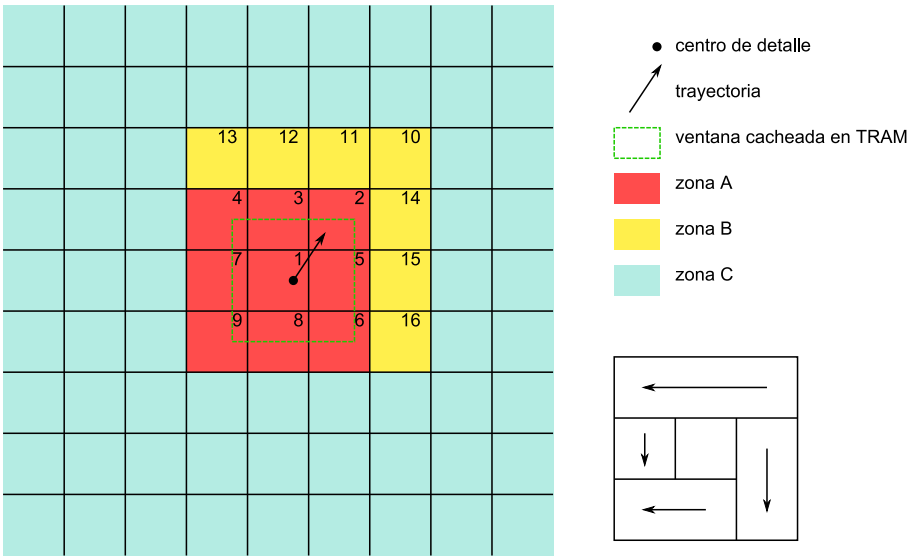


Figura 4.17: Comportamiento del algoritmo de asignación de prioridad de carga a los *buffers* de `CacheRAM` para un ejemplo de dirección de avance del centro de detalle.

`CacheTRAM`. Por este motivo, dentro del nivel tendrán la máxima prioridad de carga en `CacheRAM`.

Los *buffers* de cada zona serán a su vez ordenados por prioridad. La zona A se cargará desde el centro hacia el exterior en anillos concéntricos, de forma similar aunque no exactamente igual a como se realizaba con las teselas en TRAM.

Las principales diferencias entre las teselas en TRAM y los *buffers* en RAM son que en este último caso no es tan importante conseguir disponer de regiones rectangulares inmediatamente para el *render* y que en cambio sí es mucho más crítico cargar primero los *buffers* con mayor esperanza de vida. De hecho, debido a la lentitud relativa de estas cargas, es posible que cuando haya finalizado la carga, la información ya no sea útil debido al desplazamiento de la cámara y por consiguiente del centro de detalle.

Debido a estas diferencias, la división en cuatro brazos descrita en la actualización de `CacheTRAM` e ilustrada en la figura 4.12 se varía de forma que el brazo frontal sí comienza en el *buffer* de la esquina hacia la que avanza el centro de detalle y cubre toda la dirección frontal de la zona A. El segundo brazo cubre todos los *buffers* del lateral de avance a excepción del primero que ya se incluyó en el primer brazo. Los otros dos brazos cubren las zonas restantes, dejando el *buffer* de la esquina opuesta a la dirección de avance en el último de los brazos.

La zona B contiene sólo los dos brazos que amplian el rectángulo de la zona A en la dirección de avance del centro de detalle. Se carga primero el

correspondiente a la dirección principal de avance, comenzando por el *buffer* de la esquina y a continuación el otro brazo siguiendo el mismo orden: desde la esquina de la L hacia el extremo.

La zona C se cargará cuando las zonas A y B estén completas en RAM, ampliando la información disponible en RAM alrededor del centro de detalle aprovechando el tiempo disponible y hasta ocupar la memoria asignada para la caché en RAM. En este caso hay dos políticas posibles: cargar anillos concéntricos alrededor de la zonas A y B o ampliar la zona A y B en la dirección de avance del centro de detalle mediante conjuntos de *buffers* en forma de L de forma equivalente a cómo se gestiona la zona B. La elección entre una y otra política vendrá determinada por el tipo de movimiento de la cámara. Si éste tiene una cierta continuidad en su trayectoria, y especialmente cuando la velocidad es elevada, será más adecuada la segunda política (cargas en L), mientras que cuando la trayectoria no sea continua se consiguen mejores resultados con el uso de la primera política (cargas en anillos concéntricos).

En todas las zonas, dentro de cada brazo, al igual que sucedía con las teselas, el orden de carga sigue la dirección opuesta a la de la trayectoria del centro de detalle.

En cuanto a la prioridad entre niveles, se puede asumir una carga de todos los niveles de la textura desde abajo (detalle mínimo) hasta arriba (detalle máximo) en caso de información estática, aunque esto no siempre será así para datos dinámicos con un ciclo de actualización breve, tal y como ha se ha comentado anteriormente. Una vez decidido qué niveles se actualizan y en qué orden, existen dos alternativas a la hora de priorizar las cargas de *buffers*.

La primera opción consiste en cargar la zona A para cada nivel en el orden de niveles correspondiente, y una vez que todos los niveles gestionados tengan la zona A completamente cargada, comenzar la carga de la zona B para cada nivel en el mismo orden. Esta estrategia proporciona el máximo detalle lo antes posible, pero como contrapartida puede provocar caídas de detalle cuando se desplace el centro de detalle, especialmente si este desplazamiento es rápido.

La segunda alternativa consiste en cargar para cada nivel por orden la zona A y a continuación la zona B, de forma que se dispone de un margen de avance en ese nivel. De esta forma, no se alcanzará tan rápidamente el máximo nivel de detalle, pero una vez disponible un nivel de detalle, éste será más duradero, evitando la reducción repentina de la calidad de la textura en el *render* al desplazar la cámara.

La primera estrategia resulta más adecuada en situaciones en que la cámara se mueve a poca velocidad o cuando se necesite obtener el detalle máximo lo antes posible en zonas puntuales, en lugar de mostrar una navegación fluida. La elección depende, por tanto, de la aplicación, por lo que ambas estrategias están soportadas y es posible cambiar entre una y otra en tiempo de ejecución, para una máxima flexibilidad. En este caso, se puede permitir

```
for nivel (en el orden correspondiente)
  generar lista ordenada de buffers zona A
  for buffer
    cargar buffer
  end for
end for
for nivel (en el orden correspondiente)
  generar lista ordenada de buffers zona B
  for buffer
    cargar buffer
  end for
end for
```

Figura 4.18: Estrategia 1 para la carga de zonas entre niveles. Disponibilidad rápida.

```
for nivel
  generar lista ordenada de buffers zona A
  for buffer zona A
    cargar buffer
  end for
  generar lista ordenada de buffers zona B
  for buffer zona B
    cargar buffer
  end for
end for
```

Figura 4.19: Estrategia 2 para la carga de zonas entre niveles. Máxima estabilidad.

al usuario decidir el comportamiento o gestionarlo de forma automática en función de la velocidad de vuelo.

Los pseudocódigos de las figuras 4.18 y 4.19 ilustran las estrategias de gestión de prioridades, pero sólo son correctos en los casos en que el centro de detalle permanezca fijo. La lista ordenada de *buffers* a cargar puede variar tras cada carga de un *buffer* debido al desplazamiento del centro de detalle, por lo que las listas se reconstruirán continuamente.

Una vez llena la memoria asignada para la caché, el comportamiento sigue un algoritmo LRU clásico para la selección de *buffers* para nuevas cargas. Es importante indicar que sólo los *buffers* de la zona C están disponibles en la cola LRU, de forma que las zonas A y B de todos los niveles estarán siempre cargadas. Por este motivo es importante que la memoria reservada para la caché sea lo suficientemente amplia para albergar las zonas A y B de todos los niveles de la textura.

4.5.1. Arquitectura de la actualización asíncrona

La caché predictiva descrita en esta sección, que precarga en memoria principal los *buffers* con la información que necesitará la caché de primer nivel (CacheTRAM), se ha desarrollado siguiendo la arquitectura que se ilustra en la figura 4.20.

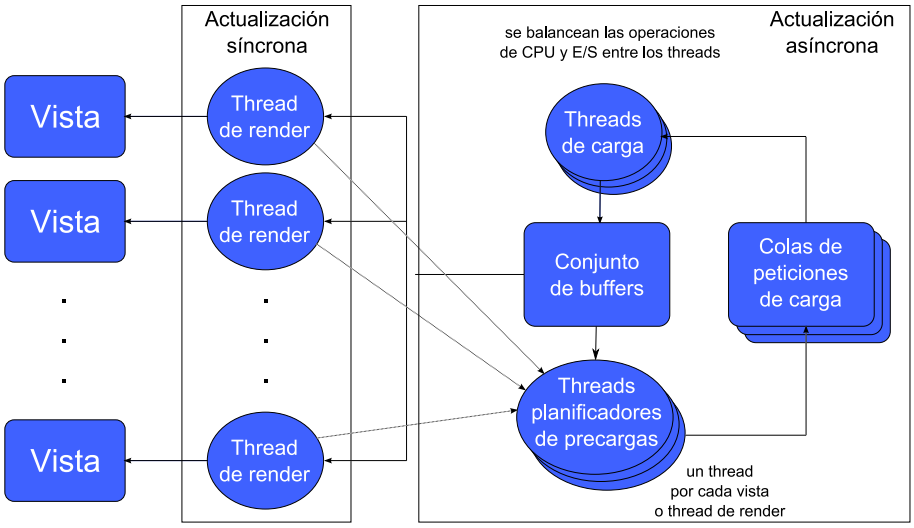


Figura 4.20: Arquitectura del sistema de actualización asíncrona de la caché predictiva.

La arquitectura que se ha diseñado permite que la caché de segundo nivel (CacheRAM) sea compartida entre diferentes vistas de la escena, cada una de las cuales tendrá su propia cache de primer nivel (CacheTRAM) en memoria de vídeo y un proceso de *render* independiente. De esta forma se consigue la máxima eficiencia tanto en el uso de memoria como en ancho de banda para la carga de información a memoria principal.

Un objeto CacheRAM podrá por tanto ser utilizado por varias vistas de la escena, cada una con su CacheTRAM correspondiente. Estas vistas o estas CacheTRAM se denominan **clientes** desde el punto de vista de la CacheRAM. Al construir un objeto CacheTRAM automáticamente se registra como cliente de la CacheRAM asociada.

Se crea un hilo de ejecución o *thread* para cada cliente, encargado de examinar el estado de su CacheTRAM para determinar cuáles son los *buffers* que necesita y con qué orden de prioridades. Estos hilos de ejecución se encargan de mantener actualizadas las listas de *buffers* a precargar, denominadas **colas de peticiones de carga** en el esquema de la figura 4.20. Los *buffers* requeridos por cada cliente y su orden de prioridad se determinan mediante los algoritmos descritos en el apartado anterior.

Para optimizar la gestión de memoria, el **conjunto de *buffers*** que componen la caché se ubica en un bloque contiguo de memoria reservado en la

inicialización, al que denominamos **arena**. Por lo tanto, el número de *buffers* disponibles es fijo y se determina en la inicialización del sistema a partir del fichero de configuración. De la misma forma, se inicializa una tabla de descriptores de *buffer* con el mismo número de elementos, donde cada descriptor apunta a una zona de la arena asociada a él.

Durante el funcionamiento del sistema, no se crean ni se destruyen descriptores ni *buffers*, sino que se reutilizan los creados en la inicialización según el comportamiento descrito y con el apoyo de ciertos metadatos adicionales: la **cola LRU** y el **mapa *hash***. Estas estructuras están inspiradas en el diseño del *buffer cache* del sistema operativo UNIX [38, 110].

La cola LRU contiene referencias a los descriptores de los *buffers* susceptibles de ser utilizados para albergar nuevas regiones del espacio virtual, ordenados de forma descendente por tiempo desde su último acceso. Cuando se necesita un *buffer* para cargar datos de una zona determinada del espacio virtual, se solicita un descriptor a la cola LRU, que será el primero de la cola.

Cada vez que se asigna un *buffer* a una zona determinada del espacio virtual, se guarda una referencia en el mapa *hash*. Este mapa proporciona un acceso rápido a las zonas del espacio virtual que están presentes en la caché y es la forma de identificar si están ausentes, para solicitar su carga.

Todo *buffer* con datos válidos estará en el mapa *hash* (los *buffers* que no tienen datos válidos serán aquellos que todavía no han sido utilizados tras la inicialización de la caché). Aquellos *buffers* que no estén bloqueados estarán además en la cola LRU.

Se considera que un *buffer* está **bloqueado** (y por tanto no se puede reutilizar para cargar una zona distinta, por lo que no está en la cola LRU) cuando forma parte de la zona A o B de un cliente, tal y como se ha descrito en la sección anterior. Puesto que la caché puede estar compartida entre varios clientes, esta protección se controla mediante un contador de referencias, que indicará el número de clientes que contienen la zona actual de ese *buffer* dentro de sus zonas A o B.

El número de *buffers* que se necesita reservar en la inicialización del sistema debe ser por tanto igual o superior al número de *buffers* que cubren las zonas A y B de todos los niveles de todos los clientes. Esta comprobación se realiza durante la inicialización para garantizar el correcto funcionamiento de la caché.

Las colas de peticiones de carga son atendidas por uno o más *threads* de carga asíncrona. El uso de varios *threads* simultáneos para las cargas puede acelerar el proceso o degradar el rendimiento de las transferencias, según los casos, dependiendo mucho de cada situación, si se trata de disco o accesos por la red, la configuración de discos o la distribución de servidores, etc.

En el caso de uso de formatos comprimidos el proceso de carga de un *buffer* se divide en dos fases: una fase inicial de descarga o *streaming* seguida de una fase de descompresión. Ambas fases se realizan de forma secuencial,

siguiendo una estructura de *pipeline*. No se comienza la descompresión hasta haber descargado el *buffer* completo. Además, ambas etapas del *pipeline* tienen unas características completamente diferentes en cuanto al uso de recursos. La primera fase de transferencia es claramente un proceso de entrada/salida, con un consumo mínimo de CPU, mientras que la segunda fase de descompresión trabaja sobre los datos ya cargados en memoria, realizando cálculo intensivo, por lo que se trata del caso opuesto. Esta división tan clara pide la paralelización de ambas fases para diferentes *buffers*, de forma que se aprovechen al máximo los recursos disponibles tanto de E/S como de CPU, balanceando los procesos para obtener el máximo rendimiento. Por este motivo el sistema permite limitar el número de *threads* dedicados a carga y el número de *threads* dedicados a descompresión (o cálculo intensivo). Mediante semáforos contadores [64] se limita el número de *threads* dedicados simultáneamente a tareas de E/S o CPU, con objeto de maximizar el rendimiento global.

Hilos de ejecución, sincronización y protección

El esquema planteado comparte diversas estructuras de datos entre diferentes *threads*, lo cual hace necesario implementar los mecanismos de protección adecuados para garantizar la integridad de los datos en los accesos concurrentes. Al problema de protección de las estructuras de datos compartidas que tiene cualquier sistema con accesos concurrentes, se añade el problema de garantizar el máximo rendimiento al menos en los hilos de ejecución correspondientes a los procesos de *render* de las diferentes vistas o clientes de la caché.

Las operaciones de los *threads* de cálculo de precargas por vista o los *threads* de carga de *buffers* no deben nunca hacer esperar a los *threads* de *render* más que lo mínimo imprescindible, sin importar que por ello se tarde más en el proceso global de la precarga de la caché.

A continuación se describe el comportamiento de los diferentes *threads* utilizados por el sistema de caché predictiva asíncrona desarrollada en este trabajo, las estructuras de datos compartidas entre ellos y los principales aspectos que se tuvieron en cuenta para conseguir un rendimiento adecuado.

En un ***thread* principal**, sin necesidad de acceso al contexto gráfico se realizarán las tareas de creación e inicialización de los objetos de la caché. En esta fase se realiza la configuración de la caché, la reserva de la arena de memoria que contendrá los *buffers* así como sus descriptores, inicialización de las estructuras como la cola LRU, el mapa *hash* y la creación de los *threads* de carga, que quedan en este momento a la espera de que haya peticiones de clientes.

Cada uno de los ***threads* de precarga**, encargados de realizar las cargas asíncronas a la caché, esperan hasta que haya peticiones por parte de algún cliente. Cuando se cumple esta condición, el *thread* recibe una señal y solicita

buffers alternativamente a los clientes registrados en la CacheRAM. De la cola de peticiones de precarga de cada cliente se retira una petición y se procesa, siguiendo una planificación *Round Robin*. Cada petición se procesa según se describe en el pseudocódigo de la figura 4.21. Cuando no existen más peticiones de ninguno de los clientes registrados, el *thread* vuelve al estado inicial de reposo a la espera de la señal que le indica que se han creado nuevas peticiones.

```
PARA CADA petición
  SI el buffer está en el mapa hash
    SI su estado es PENDIENTE
      descartar petición (en carga por otro thread)
    SINO SI el buffer está CADUCADO
      SI se consigue bloquear para carga
        cargar buffer (bloqueo)
        actualizar estado
        proteger si procede (retirar de la LRU)
      FIN-SI
    SINO
      actualizar estado del buffer
      proteger si procede (retirar de la LRU)
    FIN-SI
  FIN-SI
FIN-PARA
```

Figura 4.21: Proceso de una petición de *buffer* por un *thread* de precarga.

Las peticiones de carga tienen asociado la información acerca de la necesidad de proteger el *buffer* solicitado, lo cual sucederá cuando se trate de un *buffer* correspondiente a la zona A o B del cliente pero no cuando corresponda a la zona C.

Antes de comenzar la carga de un *buffer* es necesario bloquearlo para su carga. Este bloqueo impide que varios *threads* intenten cargar el mismo *buffer* simultáneamente.

Los ***threads* planificadores de precargas** tienen como misión mantener actualizada la cola de peticiones de carga asociada a cada cliente. Estas tareas se realizan en un *thread* asociado a cada cliente, independiente de su *thread* de *render*. Ambas tareas se separan para no afectar al rendimiento del *thread* de *render*, ya que realiza la operación más crítica del sistema. Causar retardos en ese *thread* podría afectar al tiempo de respuesta y la fluidez de la navegación y la visualización. La cola de peticiones de precarga se calcula a partir de la ubicación del centro de detalle y por tanto la zona cacheada en TRAM del cliente. Concretamente se utilizan las matrices de estado (con direccionamiento toroidal) que contienen los metadatos de

la caché para cada nivel. La CacheRAM predictiva utiliza una colección de estructuras equivalentes a las ya descritas para la CacheTRAM en la sección 4.4.3.

Cada vez que se desplaza el centro de detalle hasta el punto de cambiar la zona cacheada en TRAM se debe actualizar la cola de peticiones. Esta cola se construye mediante alguna de las estrategias o algoritmos descritos en el apartado anterior, que determinan las zonas A, B y C del cliente y el orden de carga de los *buffers* dentro de cada uno.

Los ***threads* de *render*** serán los usuarios finales de la CacheRAM predictiva, puesto que son ellos quienes tienen acceso al contexto gráfico donde se realizarán las actualizaciones síncronas que cargan la información desde RAM a TRAM. La operación desde estos *threads* es la más sencilla, simplemente se solicitan los *buffers* que se necesitan de forma inmediata para las tareas de actualización síncrona. En el caso óptimo, estos *buffers* habrán sido precargados y estarán ya disponibles en la caché, por lo que se solicitan al mapa *hash* y se accede a sus contenidos. Si no están disponibles, no se hace nada más que continuar la actualización síncrona, los otros *threads* ya descritos de la CacheRAM predictiva se encargarán de proporcionar los *buffers* lo antes posible en función de los recursos disponibles y las prioridades de cada uno.

Debido a la evidente conexión entre los diferentes *threads* y para evitar bloqueos largos al operar sobre los descriptores de *buffers* y las otras estructuras implicadas, se ha intentado minimizar las zonas bloqueadas. Esto ha supuesto un aumento en el número de *mutexes* necesarios[6], con el consiguiente riesgo de interbloqueo. Para evitar estos potenciales problemas, se ha establecido un orden estricto de bloqueo de los diferentes mutexes.

También se han utilizado variables de condición[47] para no consumir recursos en los *threads* que esperan un determinado evento ni entorpecer la ejecución de los otros *threads*. Ejemplos inmediatos son la espera de los *threads* de carga por peticiones cuando las colas están vacías o la espera a desplazamientos de la zona cacheada en TRAM para rehacer las colas de peticiones de cargas.

Para la depuración de los *threads* y su sincronización se ha utilizado el programa **helgrind**, perteneciente al conjunto de herramientas **Valgrind**[28] para depuración y análisis de rendimiento de *software*.

Algunas de las estructuras de datos en las que se ha controlado la concurrencia en el acceso desde diferentes *threads* son las siguientes: cola LRU, mapa *hash*, lista de clientes, cola de peticiones de cada *thread* de planificación de precargas, contadores de referencias de los descriptores de *buffer* o las ya mencionadas variables de condición como el número de peticiones

[6] En la implementación actual existen diez tipos de mutexes, de los cuales algunos están asociados a múltiples elementos, como *threads* o descriptores de *buffers*, por lo que el número total de instancias es en realidad muy superior.

pendientes de atender o el cambio de estado de la caché en TRAM. También se han utilizado *mutexes* asociados a los *threads* de planificación de cargas y de carga para poder detener su ejecución en un estado controlado, lo cual es necesario para determinadas operaciones como el vaciado de las cachés en los cambios de orígenes de datos.

En el siguiente capítulo se analizarán los resultados obtenidos de esta caché predictiva, realizando mediciones de fallos de caché y tiempos consumidos en cada fase. Los resultados serán comparados con la caché reactiva basada en una cola LRU que comienza la carga de los *buffers* una vez han sido solicitados por el primer nivel de caché (CacheTRAM).

4.6. Gestión de datos vectoriales

4.6.1. Arquitectura de las cachés para la gestión de datos vectoriales

Los módulos descritos hasta el momento corresponden con la visualización de información de tipo *raster*. En esta sección se tratarán las características propias del tratamiento de datos vectoriales en el sistema desarrollado.

Una de las diferencias más importantes en la visualización de datos vectoriales es que se pierde la predictibilidad, en cuanto al tiempo de actualización síncrona de una tesela, que ofrecían las cargas o transferencias de bloques de memoria de tamaño fijo entre las cachés. Las teselas de la caché en TRAM que se alimentan de los datos vectoriales se rellenan mediante el *render* a textura de la zona correspondiente.

En estos casos, la estructura en que se almacenan los datos en memoria principal no corresponde en los dos niveles de caché. El primer nivel de caché (CacheTRAM) trabaja en formato *raster*, como ya se ha descrito, mientras que la caché de segundo nivel (CacheRAM) almacena los datos en formato vectorial. Estos datos vectoriales se mantienen en memoria principal organizados en una estructura de grafo de escena (*scene graph*). La actualización de una tesela de la CacheTRAM no consiste por lo tanto en una mera transferencia de datos *raster* de una memoria a otra, sino que implica un proceso de conversión o **rasterización** de los datos vectoriales. Este proceso, a diferencia de la carga de un bloque de memoria, no depende únicamente del tamaño del bloque *raster* correspondiente a la tesela que se alimenta, sino que depende mayoritariamente de la complejidad de la escena vectorial en la zona que corresponde a esa tesela.

La visualización de estos datos vectoriales 2D sobre el terreno se realiza siguiendo la misma arquitectura descrita en las secciones anteriores para los datos *raster*, pero con ciertas particularidades que se describen en este apartado. El primer componente de la cadena mostrada en la figura 4.4 es común en ambas situaciones: CacheTRAM3D, pero en el caso de la información vectorial, se utiliza un cargador específico para el manejo de *scene*

graphs: CargadorTRAM3DRender.

Funcionamiento de los componentes utilizados con los datos vectoriales

En la inicialización, CargadorTRAM3DRender construye un *Frame Buffer Object* (en adelante FBO), que utilizará para realizar el *render* a textura de cada tesela de la caché de primer nivel. Al comienzo de cada ciclo de actualización, se guardan los estados de las matrices de transformación de modelo-vista y de proyección (MODELVIEW y PROJECTION en OpenGL) y se activa el FBO asociado al cargador, para que las siguientes operaciones de *render* tengan como destino dicho FBO en lugar de la pantalla. Al comienzo de la actualización de cada nivel dentro del ciclo, se calcula la capa de la textura 3D (o del *array* de texturas) a la que corresponde ese nivel de la textura virtual y se asocia dicha capa con el FBO para establecerla como destino del *render*. Dentro del nivel, para cada tesela que se actualiza, se establece el *viewport* correspondiente a la zona la capa de la textura 3D (o *array* de texturas) que ocupa la tesela, se borran los contenidos dentro de ese *viewport*, y se delega en la caché de segundo nivel (CacheRAM), que es la que contiene los datos vectoriales, para que realice el *render* de esa zona. Al final de la actualización, se desactiva el FBO y se recupera el *viewport* y las matrices guardadas al comienzo para restaurar el estado de OpenGL tal y como estaba antes de comenzar la actualización.

La CacheRAM utilizada en este esquema de datos vectoriales se denomina CacheRAMSceneGraph, y trabaja en conjunto con un CargadorRAM compatible. En este trabajo se desarrollaron varios cargadores con características diferentes. El que se describe en este apartado y se utilizará para las pruebas es CargadorRAMSceneGraphOSG, que está basado en la librería de grafo de escena OpenSceneGraph[18].

El papel de CacheRAMSceneGraph es bastante limitado, se mantiene principalmente para conservar la coherencia en la arquitectura del motor de texturizado. Eventualmente puede utilizarse para cambiar entre diferentes colecciones de datos (basadas en grafos de escena), pero para las tareas de *render* delega completamente en el cargador que es quien contiene la escena en sí.

CargadorRAMSceneGraphOSG contiene la escena vectorial que se proyectará sobre el terreno. Contiene un visor OSG lo más reducido posible, de forma que lo único que se hace es establecer las matrices de modelo-vista y proyección y *renderizar* la escena (el *viewport*, el borrado y la activación del FBO y la capa de textura correcta ya ha sido realizado por CargadorTRAM3DRender).

4.6.2. Optimización de los datos vectoriales

El origen de los datos vectoriales suelen ser servicios como el WFS del OGC, utilizando lenguaje GML como formato para especificar los contenidos, o ficheros procedentes de sistemas GIS (en formatos como el SHP de ESRI) o CAD (en formatos como DXF o DGN). En todos estos casos, los datos suelen estar organizados en capas, siguiendo algún criterio significativo para los usuarios, pero en absoluto están organizados adecuadamente y mucho menos optimizados para su visualización eficiente de forma interactiva.

Por este motivo, se hace imprescindible una fase de preproceso de esta información vectorial antes de utilizarla en el motor de texturizado.

Los objetivos de este preproceso de la información vectorial se pueden resumir en los siguientes:

- Mejora del rendimiento de la actualización de las teselas de la caché de textura.
- Mejora de la predictividad en el tiempo de actualización de dichas teselas.
- Posibilidad de variación del aspecto dependiendo de la situación de la cámara 3D.
- Posibilidad de variación del aspecto dependiendo de la escala de visualización.
- Ajuste de la calidad de visualización en función de la carga del sistema y por tanto del tiempo disponible para la actualización síncrona.

El resultado de este preproceso será un grafo de escena con la estructura adecuada para una visualización interactiva eficiente, cumpliendo con las características requeridas como puede ser la variación de la representación de los datos vectoriales.

Estos procesos de optimización se pueden realizar de varias formas, combinando diversas técnicas, según la calidad, el rendimiento y las características que se deseen obtener.

Algunas de las técnicas desarrolladas en este trabajo se describen más adelante en esta sección. Pueden llegar a ser procesos notablemente costosos en tiempo de ejecución, por lo que a priori se plantean dos formas de realizarlos:

- Preproceso *off-line*. Las colecciones de datos se preparan previamente para su incorporación al sistema de visualización en tiempo real. La preparación de los datos se realiza por tanto como un proceso anterior a la ejecución de la aplicación de visualización y como resultado de este proceso, una nueva colección de datos se añade al catálogo disponible

por dicha aplicación para su visualización interactiva. El tiempo de proceso no es excesivamente crítico, puesto que se realiza una única vez al incorporar cada nueva capa de información al sistema.

- Preproceso "al vuelo". Cuando los datos que se visualizan son proporcionados por el usuario durante la ejecución de la aplicación de visualización, se hace necesario realizar el proceso de optimización de los datos como una fase de la carga de los datos. En este caso el tiempo de proceso cobra más importancia, puesto que será tiempo de espera para el usuario que ha solicitado su visualización.

Por lo tanto, los procesos realizados serán diferentes en cada una de las dos situaciones descritas, en función del tiempo que se considere aceptable en cada caso. Es recomendable, siempre que sea posible, realizar el preproceso *off-line*. En caso de no ser posible, y que este proceso se realice durante la visualización, la optimización deberá ser menos agresiva para reducir el tiempo de espera del usuario. A continuación se describen las principales operaciones realizadas para mejorar el rendimiento de la actualización de la caché y posibilitar el cumplimiento de los objetivos enumerados para la visualización interactiva de los datos vectoriales. En el capítulo 5 se presentarán los resultados cuantitativos obtenidos de la aplicación de dichas técnicas sobre una escena vectorial de prueba.

Organización jerárquica

En el *render* de escenas complejas, uno de los principales mecanismos de optimización, esencia de la estructura de datos de un grafo de escena (*scene graph*), es el proceso de filtrado denominado *view frustum culling*, que descarta aquellos elementos de la escena localizados fuera de la zona visible.

La comprobación de si un elemento de la escena está visible se realiza comparando su volumen envolvente contra el volumen (o *frustum*) de visión de la cámara. Por motivos de eficiencia en los cálculos, estos volúmenes envolventes se aproximan mediante estructuras que facilitan dichos cálculos. Habitualmente se utilizan esferas (*bounding sphere*) o cajas alinedas con los ejes (*axis aligned bounding box* o AABB). Todo nodo del grafo de escena tiene asociado un volumen envolvente. Cuando varios elementos se agrupan, el volumen envolvente del grupo se calcula como el volumen mínimo que contiene los volúmenes envolventes de sus nodos hijos, en una operación de unión de volúmenes envolventes.

Por este motivo, para maximizar la eficiencia de este proceso, la escena debe tener una organización jerárquica adecuada. Con una jerarquía completamente plana, como suele ser el caso de la mayoría de los formatos de CAD y GIS, que no utilizan el concepto de grafo de escena sino más bien una colección de entidades de dibujo, es necesario recorrer todos y cada uno de

los elementos atómicos de la escena para verificar si están dentro del volumen de visión. Agrupando esos elementos en una jerarquía con criterios de proximidad espacial, al hacer un recorrido del grafo de escena desde la raíz, se pueden descartar grupos que potencialmente contendrán muchos elementos en una sola operación, sin necesidad de verificarlos uno por uno.

En el caso que nos afecta, puesto que se realizarán *renders* a una textura cuyo espacio total está formado por niveles divididos en teselas cuadradas de igual tamaño y doble número de teselas en cada nivel consecutivo de detalle más fino, la estructura de árbol cuaternario o *quadtree*[130] resulta idónea. Se harán coincidir exactamente las teselas con los nodos del *quadtree*.

La profundidad óptima del árbol cuaternario para organizar los datos vectoriales de la caché de una GeoTextura se puede calcular fácilmente a partir del tamaño de la textura virtual y del tamaño de la tesela de actualización síncrona en TRAM. La ecuación 4.3 calcula el número de niveles en profundidad del árbol óptimo (p), donde s es el tamaño lado mayor de la textura virtual en *texels* y t el tamaño de lado de la tesela de actualización síncrona también en *texels*.

$$p = log_2\left(\frac{s}{t}\right) \tag{4.3}$$

Continuar la subdivisión de la geometría a partir de ése nivel es contraproducente, puesto que los contenidos de esas hojas se van a utilizar siempre en bloque, y los niveles siguientes sólo aumentan la complejidad del grafo de escena, el tiempo de recorrido del mismo y la cantidad de memoria utilizada.

En muchos casos, la estructura de árbol cuaternario se utiliza dividiendo y clasificando los elementos de la escena de forma que finalmente sólo los nodos hoja del árbol contienen la geometría. En nuestro caso, la geometría se almacena en los nodos del árbol a cualquier nivel, según su tamaño, y por tanto el número de teselas que ocupa.

En el capítulo 5 se analiza el impacto de esta organización jerárquica en la eficiencia del *render* a textura que constituye la fase de actualización de la textura con datos de origen vectorial.

Niveles de detalle

Con la organización jerárquica en un árbol cuaternario adaptado a la estructura de teselas de la caché en TRAM, descrita en el apartado anterior, se puede llegar a mejorar el tiempo de generación de teselas de forma muy drástica. La mejora en el tiempo de *render* es proporcional a la cantidad de datos vectoriales que se manejen.

Sin embargo, estas mejoras en el tiempo de *render* se centran especialmente en las teselas de los niveles de mayor detalle de la textura virtual. El motivo es que el tener un detalle muy fino supone que se centran en una zona geográfica reducida y el proceso de *view frustum culling* apoyado por la

estructura de árbol cuaternario descarta rápidamente todos los elementos de la escena situados fuera de la zona correspondiente a la tesela actualizada.

Son los niveles de menor detalle de la textura los que cubren toda el área geográfica y por tanto necesitarían a priori el *render* de todos los datos vectoriales para actualizar la tesela. Sin embargo, la mayoría de los datos vectoriales presentes en la escena no son apreciables a esa escala, por lo que realmente no haría falta toda esa información.

En este caso existe una solución muy adecuada: el uso de unos nodos grupo especiales del grafo de escena, denominados "niveles de detalle" o nodos LOD (*level of detail*). Mediante estos nodos LOD se puede definir un intervalo de tamaño en *texels*, fuera del cual se descartan del *render* los elementos vectoriales que no aporten información apreciable en la imagen final.

Esta técnica de optimización no sólo permite reducir el tiempo de *render*, sino que al mismo tiempo ayuda a aproximar el de las diferentes teselas. Esto es muy importante, si tenemos en cuenta que el proceso de actualización síncrona está limitado en tiempo. En dicho proceso se actualizan las teselas invalidadas de la caché de primer nivel por orden de necesidad, tal y como se ha descrito en la sección 4.4, mientras haya tiempo disponible en el fotograma actual. La existencia de teselas cuyo tiempo se dispara respecto a las demás provocará que eventualmente se desborde el tiempo de *render* o en el mejor de los casos requerirá unos márgenes de seguridad que afectarán gravemente a la velocidad de actualización.

En el optimizador desarrollado en este trabajo se ha tomado la decisión de diseño de utilizar nodos LOD como los propios nodos del árbol cuaternario. De esta forma se aligera y se reduce la complejidad del grafo de escena resultante. Se ha utilizado el tamaño del volumen envolvente de los nodos proyectado en espacio pantalla para determinar los elementos de la escena que se descartan del *render*.

El uso de los nodos LOD se puede combinar con el ajuste dinámico de carga de actualización. En situaciones de peligro de desbordamiento del tiempo de cuadro, se puede ayudar a reducir el tiempo de actualización de las teselas mediante la variación del tamaño mínimo de los elementos dibujables del grafo de escena. Esto es tan rápido como variar un parámetro numérico en los nodos LOD previamente a la fase de dibujado de la escena.

Una técnica para evitar el desbordamiento del tiempo de actualización consiste en guardar en cada nodo del árbol cuaternario un registro del tiempo de *render* de la tesela correspondiente a ese nodo. De esta forma, las teselas a actualizar se planifican en función del tiempo disponible conociendo a priori una estimación del tiempo de *render* que necesita cada una. Esta técnica se puede combinar con el ajuste dinámico de los márgenes de los nodos LOD para igualar los tiempos de actualización de teselas o para simplificar aquellas teselas que plantean un problema debido al tiempo de *render* excesivo que requieren.

En el caso de ajuste de los datos descartados en los nodos LOD debido a una situación temporal de carga alta, se facilita el futuro refinamiento de los datos mediante el establecimiento de un período de caducidad en primer nivel breve para esas teselas, aunque el segundo nivel de caducidad sea el habitual. Tal y como se describió en la sección 4.4.4 esto producirá el efecto de que la versión más simplificada de la tesela se utilizará normalmente hasta que se disponga de tiempo de actualización para generar una versión más detallada que la sustituya.

Generalización cartográfica

Los nodos LOD no sólo permiten descartar elementos vectoriales por su tamaño reducido en la textura, sino que también son el mecanismo idóneo para mostrar representaciones diferentes de dichos elementos según la escala de visualización, implementando el concepto de **generalización cartográfica**.

Por ejemplo, una carretera se podría representar mostrando todos sus carriles en una vista cercana y como una línea en una vista lejana. Una población se podría mostrar como los contornos de manzanas, bloques o edificios en una vista cercana y sustituirse su representación por un círculo en una vista más lejana.

Estos comportamientos se controlan perfectamente mediante el uso de los nodos LOD, sin un coste adicional apreciable en el rendimiento de la visualización.

La generalización cartográfica digital, según describen McMaster y Shea [111], incluye un conjunto de procesos como la simplificación, fusión, eliminación, refinamiento, exageración o el desplazamiento de los elementos representados. Algunos de estos procesos son automatizables en mayor o menor medida y otros poseen unas cualidades altamente subjetivas, en muchos casos se trata incluso de decisiones artísticas o estéticas. De hecho, Max Eckert ya definía en 1908 los mapas generalizados como obras de arte clarificadas por la ciencia [67]. Arthur H. Robinson, en 1960 declaraba que sería imposible establecer un conjunto de reglas que definiesen exactamente los procesos de generalización [126]. Robinson sospechaba que la generalización se mantendría para siempre como un proceso intrínsecamente creativo y escaparía por tanto a la tendencia actual a la automatización. En palabras de Edward Imhof, las máquinas no poseen juicio geográfico ni sentido de la estética, por lo que los contenidos y la creación gráfica de los mapas se reservan para el trabajo crítico del cartógrafo [95].

Existen numerosos trabajos en la automatización[7] de los procesos de

[7] Aunque se haga referencia a la automatización, es necesario suministrar ciertos parámetros de operación y una supervisión de los resultados, ambas tareas realizadas por una persona con los conocimientos cartográficos necesarios, por lo que más que más que procesos automáticos se deberían denominar semi-automáticos o supervisados.

generalización cartográfica digital [101], que resuelven problemas concretos de forma aislada, pero el tratamiento holístico que necesitaría una solución completa al problema parece estar todavía fuera de nuestro alcance.

El esquema de grafo de escena propuesto en este trabajo para los datos vectoriales permite la implementación de todos estos procesos, a través de los nodos LOD ya descritos, que determinan cuál de las vistas se deberá usar en cada caso, combinados con la información sobre posición y orientación de la cámara 3D. Los trabajos de generalización, no obstante, deberán estar realizados previamente en el origen de los datos vectoriales, es decir, la fuente de datos del CargadorRAM debe suministrar esta información.

En algunos casos se podrían añadir mecanismos de automatización de procesos de generalización dentro del cargador, como se ilustra en la figura 4.2, mediante la implementación de algoritmos existentes para estas tareas.

En el caso de incluir la información de generalización en los datos en origen se hace necesario el uso de formatos que soporten estas características, como el GML al que se podría acceder a través de servicio WFS, ambos estándares definidos por el OGC.

Grosor de los trazos

Durante este desarrollo se han detectado algunos problemas en el *render* a textura de información vectorial con las estructuras basadas en puntos y líneas, relativos al grosor de los trazos y de los puntos. Estos problemas no se producen con las estructuras basadas en formas poligonales, puesto que el concepto de grosor del trazo o punto no existe en estos casos.

En primer lugar, la definición del grosor de un trazo se puede realizar de distintas maneras, según el espacio vectorial que se considere:

- Espacio mundo. El grosor de los trazos se determina en unidades de medida en espacio mundo (habitualmente en metros). De esta forma se puede adecuar el grosor al tipo de información que se presenta, y por tanto a la densidad de trazos que puede suponer. Se evita así la aglomeración de trazos excesivamente gruesos que imposibilitarían la correcta visualización de la información. La desventaja de esta definición del grosor es que al alejarse de la zona visualizada se dejarán de ver los trazos (haciendo difícil localizar dónde hay o no elementos vectoriales) y al acercarse su tamaño llegará a ser excesivo y llenará la pantalla.

- Espacio pantalla. El grosor de los trazos se determina en unidades de pantalla, i.e. en pixels. De esta forma se garantiza que siempre sean visibles, independientemente de la distancia desde la que se observen. Sin embargo, al aumentar dicha distancia, los trazos se fundirán impidiendo su correcta visualización.

El segundo caso no se puede implementar exactamente en el motor de textura desarrollado, puesto que los vectores se *rasterizan* sobre la textura que luego se proyecta sobre el modelo 3D del terreno que será visto en perspectiva. Sin embargo sí se puede calcular el grosor de los trazos para que su tamaño una vez aplicada la textura se aproxime a un tamaño determinado en espacio pantalla.

Una implementación directa (y muy eficiente) del sistema utilizaría las primtivas de OpenGL `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP` o `GL_LINE_LOOP`. Sin embargo, esto tiene el problema de que OpenGL establece límites al grosor de los trazos. En las configuraciones *hardware* utilizadas en este trabajo, el grosor de línea máximo se situa en 10 pixels y el mínimo en 1 cuando no se utiliza suavizado de líneas y 0,5 cuando se utiliza.

El problema es incluso mayor si tenemos en cuenta que en el caso que nos interesa no se está realizando un único *render* a pantalla, sino que se realizan múltiples *renders* a la textura que luego se proyectará sobre la geometría del terreno visualizada en pantalla. Para mantener la coherencia en el grosor de línea, se adapta en cada nivel de la textura, de forma que cada nivel con mayor detalle utiliza el doble de grosor. El problema, en una vista no cenital es que los límites de OpenGL al grosor, pueden provocar cambios de grosor visibles a lo largo de una línea, como se puede observar en la figura 4.22.

Figura 4.22: Problemas en el grosor de línea debido al límite de OpenGL en cada nivel de detalle de la textura.

Este problema ocurre por igual tanto si se define el grosor en espacio mundo como en espacio pantalla/espacio textura.

En el caso de que las primitivas sean puntos o líneas, además de los límites que OpenGL establece para su grosor, existe un problema adicional cuando la textura no se mapea sobre la geometría manteniendo las proporciones (i.e., se realiza un escalado no uniforme): la representación de los puntos (sean círculos, cuadrados o cualquier otra forma) y las líneas se verá deformada al mapearse sobre la geometría del terreno.

Este hecho de que no coincidan las proporciones en espacio textura y en espacio mundo es bastante habitual, puesto que el tamaño de textura se adapta a una potencia de dos en ambos lados para facilitar el trabajo del motor de texturizado y en especial el teselado del espacio de la textura virtual.

La solución a estos problemas pasa por convertir las primitivas de tipo línea o punto en polígonos con determinado tamaño en espacio mundo, de forma que se evitan las limitaciones de grosor de OpenGL y se crean los elementos con las proporciones correctas. Esto dificulta, sin embargo, el mantener un tamaño fijo en espacio pantalla, por lo que habitualmente se trabajará con los trazos en espacio mundo.

Un aspecto a tener en cuenta a la hora de realizar la conversión de líneas a polígonos es la importancia de respetar la continuidad de las líneas o la conexión entre ellas. Si no se tiene en cuenta este aspecto, se pueden producir situaciones como la que se ilustra en la figura 4.23. Esto se puede resolver fácilmente considerando que los segmentos de las primitivas de tipo polilínea deben permanecer unidos, como se ha hecho en la implementación utilizada en las pruebas. En la figura 4.24 se muestra cómo estos segmentos mantienen la continuidad perfectamente. Para ello se han considerado los puntos de intersección de las rectas correspondientes a los bordes de los segmentos contiguos, según el grosor de trazo.

Para preservar la conexión entre líneas sueltas se podría comprobar si un vértice está compartido por más líneas, pero esto aumenta la complejidad de los cálculos y por tanto su coste en tiempo, especialmente si son más de dos líneas las que comparten vértice. En estos casos sería necesario teselar más la geometría poligonal para que las líneas encajen correctamente.

Representación dependiente de la vista

Además de los datos espaciales que se describen, el origen de datos puede suministrar atributos asociados a entidades geográficas. Un ejemplo claro de ello pueden ser las etiquetas de texto añadidas a los mapas para identificar esas entidades por su nombre.

Entre las operaciones de generalización cartográfica descritas por McMaster y Shen [111] se incluyen diez transformaciones espaciales y dos transformaciones de atributos (clasificación y simbolización). Todas estas transfor-

Figura 4.23: Líneas sueltas convertidas a polígonos sin tener en cuenta su continuidad o la conexión entre ellas.

Figura 4.24: Líneas sueltas convertidas a polígonos teniendo en cuenta la continuidad y las conexiones entre ellas.

maciones están orientadas hacia la generación de un mapa bidimensional y con una orientación fija, que habitualmente será hacia el norte.

Sin embargo, en una aplicación de visualización interactiva de información geográfica sobre un modelo 3D de terreno como la desarrollada en este trabajo, la vista del terreno es variada arbitrariamente a voluntad del usuario. En este caso, los elementos no espaciales como los textos o iconos situados sobre el terreno para complementar la información sobre los elementos es-

paciales se deben orientar en función de la posición de la cámara 3D para facilitar su lectura por parte del usuario.

Por este motivo, determinados elementos del grafo de escena de los datos vectoriales se consideran como "rotables" para alinearlos con la cámara. En estos casos, se asocia a los nodos del árbol cuaternario correspondientes a las teselas de la caché un registro de la orientación de la cámara en el momento de la generación de la tesela. El proceso de actualización de dichas teselas detecta cuándo la diferencia de orientación de la cámara actual respecto a la de la última actualización supera un umbral determinado y en ese momento ordena la recarga de la tesela mediante el mecanismo de doble caducidad ya descrito en la sección 4.4.4.

Además de la orientación de la cámara respecto al norte, se puede considerar el ángulo de cabeceo para que cuando la vista sea más próxima a la horizontal se desactive la proyección de textos e iconos (elementos rotables) sobre la textura y cuando se aproxime más a la cenital se active dicha proyección. Esto se puede combinar con una visualización de esos elementos rotables en forma de cartel orientado hacia la pantalla (*billboard*) situado a cierta altura sobre el terreno cuando se desactive su visualización sobre la textura.

Suavizado de líneas y contornos de polígonos

Debido a la naturaleza discreta de las imágenes manejadas en el ordenador, ya sea la propia pantalla, una textura o cualquier memoria intermedia, en el *render* se produce un efecto indeseado denominado *aliasing*. Este efecto ha sido descrito en la sección 2.6.1 para el caso del mapeado de texturas, pero resulta incluso más apreciable en el contorno de las líneas y polígonos.

Para solucionarlo, se utilizan las denominadas técnicas de *antialiasing*, que a grandes rasgos se basan en tomar varias muestras para cada pixel que posteriormente se combinan para calcular el valor final de color para ese pixel. Estas técnicas están implementadas en el *hardware* gráfico, pero no sin un cierto coste en el rendimiento del *render*.

En el caso que nos atañe, de texturizado de información vectorial, el *aliasing* del contorno de las primitivas geométricas proyectadas sobre la textura se suma al que se produce por efecto del mapeado de texturas, por lo que es especialmente importante utilizar las técnicas de *antialiasing* tanto en la textura como en la información vectorial *renderizada* a las teselas de la caché de esa textura.

Las técnicas de *antialiasing* relativas a la aplicación de la textura, que consisten en el uso de diferentes filtros para esa textura (figuras 4.25 a 4.27), ya han sido descritas en secciones anteriores. Para las texturas de origen vectorial que se tratan en esta sección es altamente recomendable utilizar el filtro anisotrópico (figura 4.27) de la textura con tantas muestras como permita el *hardware* con un rendimiento aceptable.

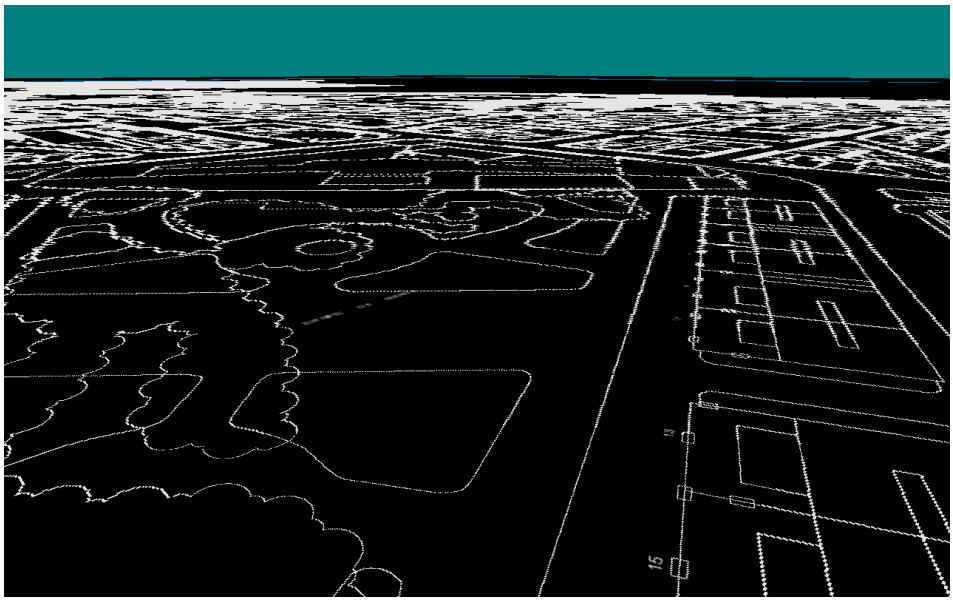


Figura 4.25: Capa vectorial proyectada sobre textura. Filtrado por proximidad.

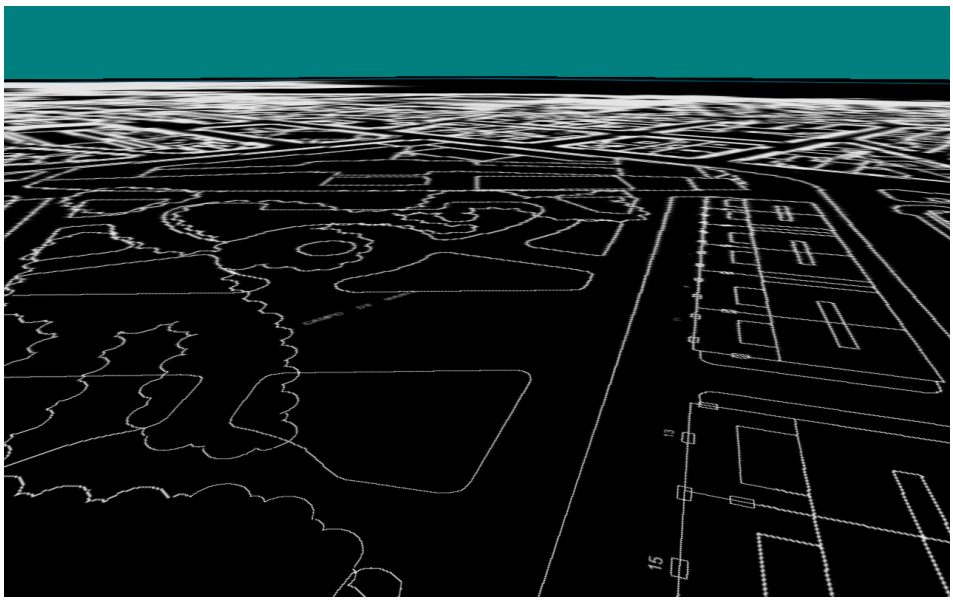


Figura 4.26: Capa vectorial proyectada sobre textura. Filtrado trilineal.

OpenGL proporciona mecanismos para el suavizado de puntos, líneas y polígonos, como se describe en el libro rojo [43] (págs. 247–260) y en el documento de especificación [135].

En el caso de líneas y puntos se suele utilizar una técnica que calcula la proporción del área del *fragment* cubierta por el punto o la línea y la multi-

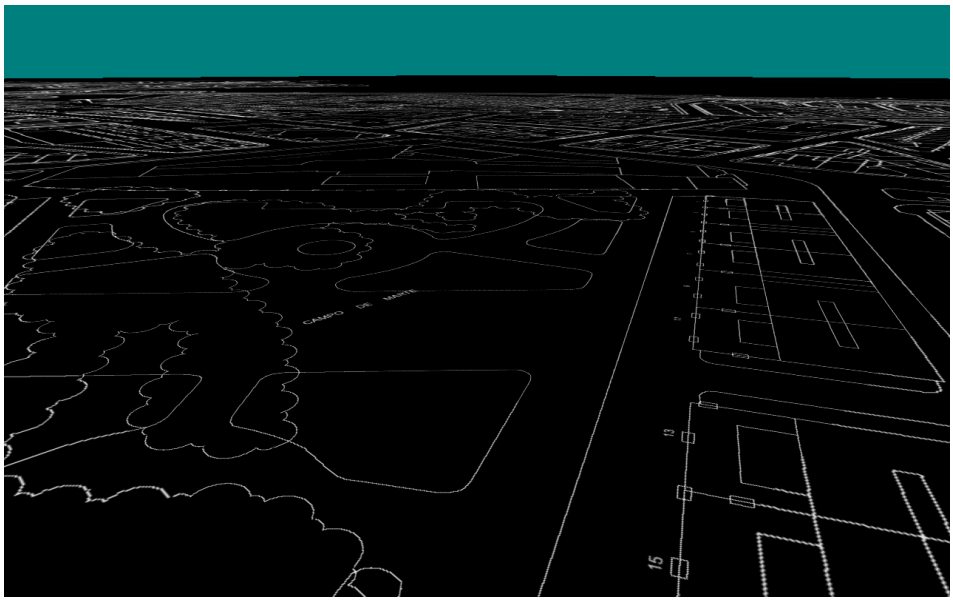


Figura 4.27: Capa vectorial proyectada sobre textura. Filtrado anisotrópico con 64 muestras.

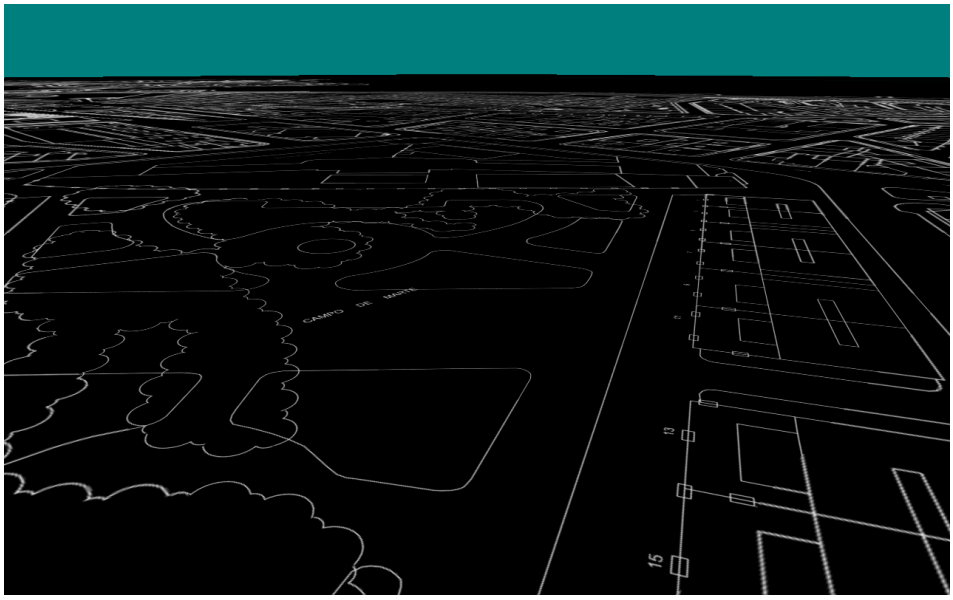


Figura 4.28: Capa vectorial proyectada sobre textura. Filtrado anisotrópico con 64 muestras y suavizado de líneas (*alpha blending + line smooth*).

plica por el valor del canal alfa de ese *fragment*. El cálculo de esta cobertura es complejo y puede variar entre implementaciones de OpenGL, el único control que se tiene es suministrar mediante la llamada `glHint()` la preferencia

en favor de calidad (`GL_NICEST`) o rendimiento (`GL_FASTEST`). El uso de esta técnica requiere la activación del suavizado, que se puede realizar independientemente para puntos (`GL_POINT_SMOOTH`) y líneas (`GL_LINE_SMOOTH`), y el modo de fundido a partir del canal alfa (*alpha blending*) de la misma forma que cuando se dibujan elementos transparentes. Esta técnica del cálculo de cobertura se puede utilizar también con polígonos (`GL_POLYGON_SMOOTH`), aunque no se suele realizar debido a que para la visualización correcta cuando los polígonos se solapan en espacio vista es necesario ordenarlos en profundidad antes de enviarlos al *render*.

Otra técnica disponible para el suavizado de puntos y líneas, y la más recomendable en caso de polígonos, es la denominada *multisampling*. Esta técnica consiste en tomar diversas muestras para cada *fragment*, distribuidas por el área de dicho *fragment* y realizar los cálculos de *render* para cada una de ellas.

No obstante, esta segunda técnica no ha podido ser implementada en el motor de GeoTextura. Aunque existe una extensión de OpenGL para utilizar *multisampling* con los FBOs, no proporciona un mecanismo para trabajar con texturas como destino del *render*, sólo *render buffers* [36].

En la sección 5.4 del capítulo de resultados se detallan las diferencias de rendimiento obtenidas en las pruebas con y sin suavizado de líneas.

4.7. Escalabilidad

Cuando la textura manejada por el sistema alcanza un tamaño muy grande en número de *texels*, surgen ciertos problemas que afectan tanto al rendimiento como a la calidad obtenida y a los recursos necesarios para gestionarla.

1. El uso de memoria se dispara debido al elevado número de niveles de la pila que se almacenan.

2. El tiempo de actualización se eleva, debido igualmente al número de niveles de la pila que hay que mantener actualizados.

3. La precisión numérica (32 bits) con la que trabaja el sistema gráfico es insuficiente para direccionar correctamente el espacio de la textura virtual.

Los dos primeros problemas se pueden solucionar aumentando los recursos de *hardware* y en el peor de los casos afectarán en mayor o menor medida al rendimiento del sistema.

En cambio, el problema de precisión numérica es mucho más grave. Por una parte, no se puede superar ese obstáculo con la gama de *hardware* gráfico que utilizamos, y por otra parte la calidad se degrada absolutamente, haciendo que el sistema no sea usable en los niveles de detalle más fino.

Uno de los casos de ejemplo que se ha planteado anteriormente consiste en una textura planetaria con resolución submétrica. En la figura 4.29 se ilustra el efecto producido por la textura de gran tamaño utilizada como ejemplo cuando la cámara se acerca a los detalles más finos. Esta textura tiene 27 niveles y en la imagen se representan los niveles más altos de la pila (rojo, cian, verde, azul, amarillo y magenta, de mayor a menor detalle).

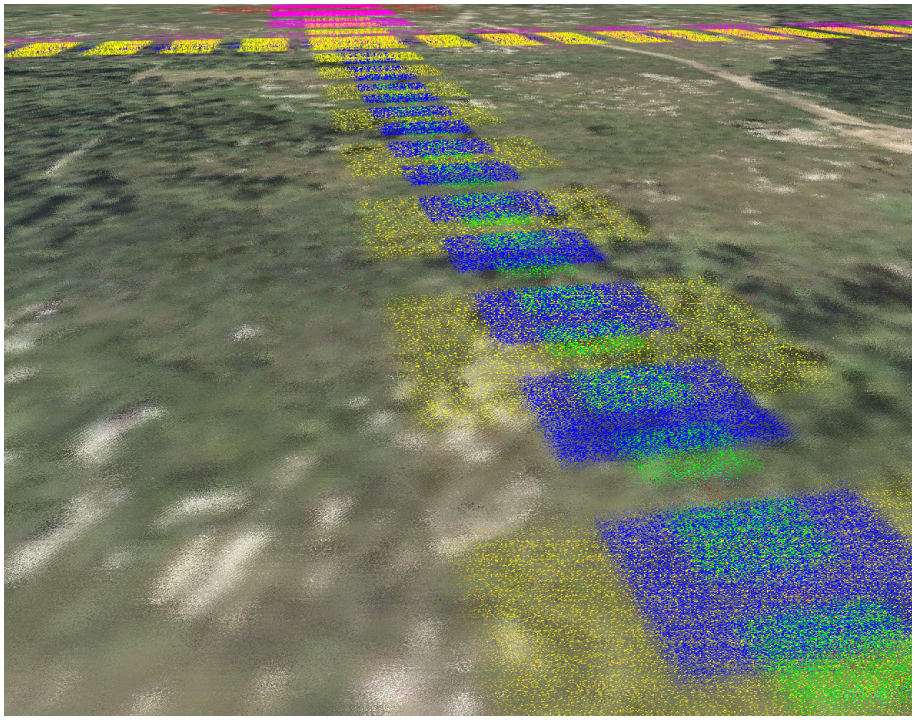


Figura 4.29: Aspecto visual de los niveles de mayor detalle de la textura cuando se supera el límite de precisión manejado por el *hardware* (32 bits).

El problema mostrado en la figura se produce porque las coordenadas de textura sufren errores de redondeo que las desplazan de la verdadera posición que deberían ocupar. Puesto que la precisión disponible en *hardware* no se puede aumentar, la única solución es reducir el área direccionada para que sea abarcable con un margen de error adecuado.

Las coordenadas de textura se almacenan en números representados en formato de coma flotante de precisión simple, según el estándar ANSI/IEEE 754-1985. Este formato de precisión simple utiliza 32 bits, de los cuales destina 1 al signo, 8 al exponente y los 23 restantes a la mantisa. Por este motivo, no podremos direccionar correctamente más de 23 niveles, es decir, el nivel más alto que se permite direccionar completamente será el nivel 22.

En los niveles superiores, se reducirá el área direccionable a un subconjunto de la extensión geográfica total de ese nivel. Esto es perfectamente factible puesto que cuando se está apreciando el detalle de niveles tan altos,

no se tendrán a la vista las zonas exteriores al área indicada.

Al mismo tiempo, para solucionar los problemas derivados del número de niveles manejados, se utilizará el concepto de **pila virtual**. No se mantendrá toda la pila en memoria, sino sólo un subconjunto de la misma.

Por lo tanto, se plantean dos soluciones complementarias para resolver los problemas descritos: reducir el número de niveles cacheados y reducir el área direccionada.

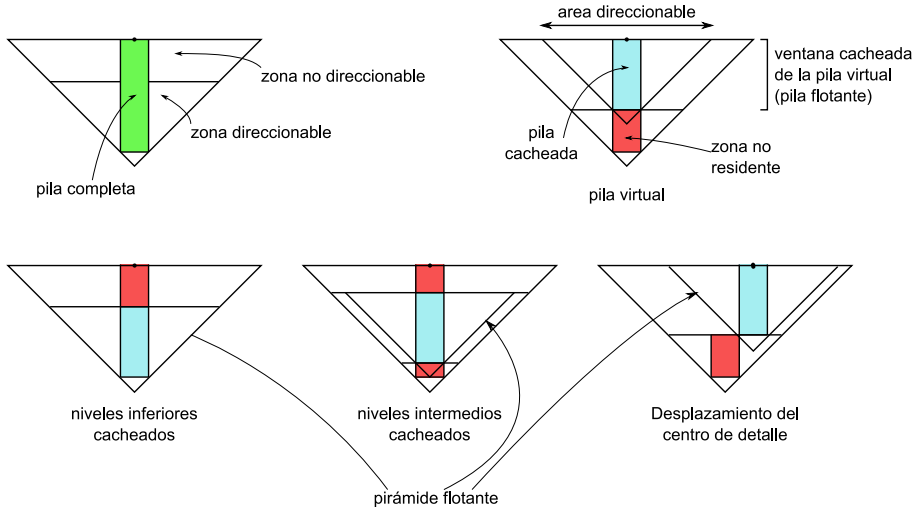


Figura 4.30: Estructura de pila virtual. Se cachea una ventana de niveles de la pila (pila flotante).

La figura 4.30 ilustra la estructura de pila virtual que se plantea. Se definirá una ventana de niveles cacheados dentro de los de la pila completa que correspondería a la resolución en *texels* de la textura virtual que se está manejando. Para evitar confusiones con la ventana bidimensional cacheada en cada nivel de la pirámide virtual, esta ventana de niveles cacheados en TRAM se ha denominado **pila flotante**, evitando de esta forma el uso del término "ventana".

El usuario será responsable de indicar en cada fotograma cuál es el máximo nivel que se desea disponible, para que la caché se reestructure adecuadamente para albergar la información necesaria. La selección de la cima de la pila (límite superior de la pila flotante) dentro del espacio virtual y la selección del área direccionable es un trabajo adicional para el cliente de GeoTextura, pero no supone en ningún caso acoplamiento con la geometría. Estas tareas se deberían realizar junto con la selección del centro de detalle, como se describe en el apéndice C.

Para minimizar las transferencias de memoria, se utilizará un direccionamiento circular (ver figura 4.31) de la textura 3D, siguiendo la misma filosofía que el direccionamiento toroidal del espacio 2D de cada nivel de detalle de la textura. En este caso la ventana de niveles cacheados se desplaza a

lo largo de la pila virtual, manejando los conceptos de desplazamiento local, desplazamiento global y tamaño de ventana igual que se hacía en los mapas 2D de teselas de textura.

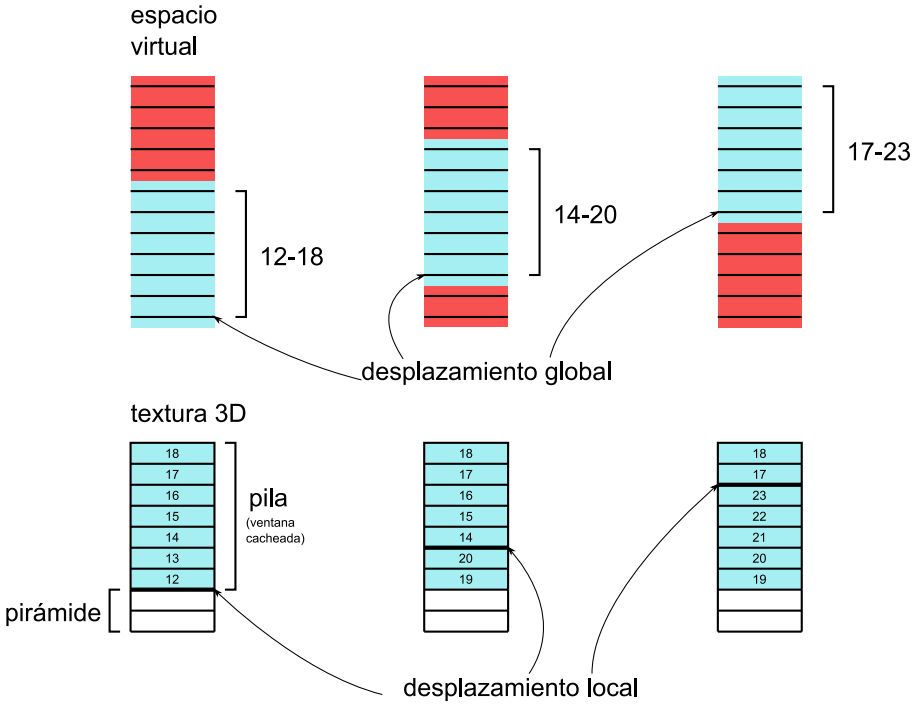


Figura 4.31: Organización en memoria física de los niveles cacheados de la pila.

Asociado a la pila flotante está el concepto de **pirámide flotante**, ilustrado también en la figura 4.30. La selección de la posición de la pila flotante junto con el área direccionable determinan la posición de la pirámide flotante que es la zona de la pirámide virtual que se manejará en la GPU.

En el apéndice D se describen las modificaciones que se deben realizar al motor de texturizado para solucionar los problemas de escalabilidad mediante la estrategia descrita.

Capítulo 5

Resultados

Tras el diseño del motor de texturas virtuales dinámicas, descrito en el capítulo anterior y completado en los apéndices, en este capítulo se realizará un análisis cualitativo y cuantitativo de los resultados obtenidos en la implementación desarrollada durante este trabajo. Para ello se han analizado diferentes situaciones y conjuntos de datos de prueba, con el objetivo de cubrir adecuadamente la variedad de posibles escenarios a los que se puede enfrentar el sistema propuesto.

Como demostrador básico del funcionamiento del sistema de texturizado, se ha desarrollado una aplicación denominada **Earthfly**, mostrada en la figura 5.1. Esta aplicación permite cargar cualquier textura virtual o combinación de éstas y visualizarla sobre una geometría plana o sobre un modelo esférico, útil cuando se trabaja a escala planetaria en coordenadas geodésicas. Earthfly, permite además visualizar información adicional, como el estado de las diferentes cachés (útil tanto para la depuración como para el análisis del funcionamiento del motor) y variar en tiempo de ejecución algunos parámetros del sistema, como el tipo de filtrado, numero de muestras, o límite de tiempo de actualización síncrona.

Las bases de datos utilizadas para la validación del sistema y las pruebas de rendimiento se describen a continuación.

Planeta Tierra. Blue Marble 2004. Conjunto de datos tomados por el NASA Earth Observatory correspondientes a los doce meses del año 2004. Cada una de las imágenes tiene una resolución de 131072×65536 pixels (8 Gigapixels) con tres canales de color de 8 bits (RGB888), lo cual resulta en 24 GBytes por imagen, 288 GBytes en total sin compresión. Las imágenes están representadas en coordenadas geodésicas y cubren completamente el planeta Tierra.

Galicia. SIGPAC. Conjunto de datos correspondiente al área geográfica de Galicia, procedente de las imágenes del Sistema de Información Geográfica de Parcelas Agrícolas (SIGPAC), perteneciente al Ministe-

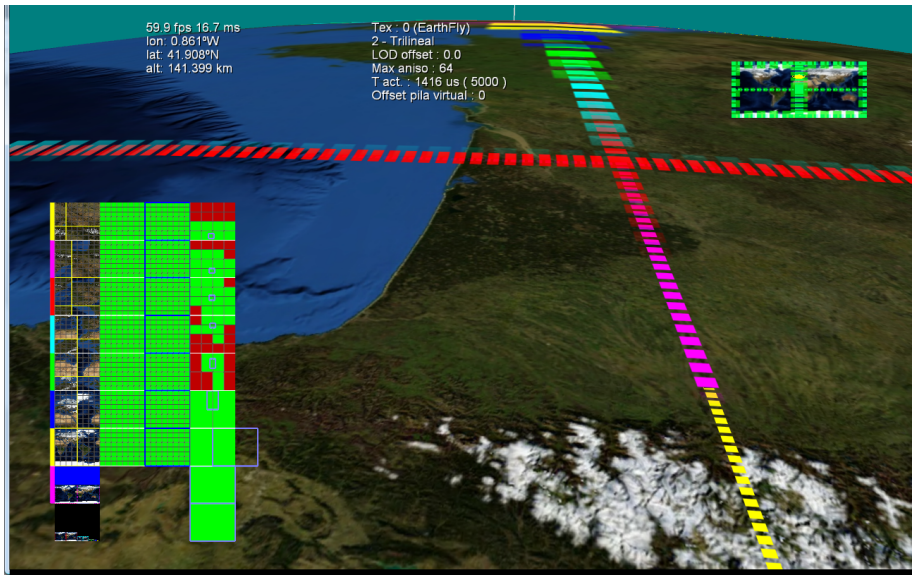


Figura 5.1: Herramienta para demostración y análisis del motor de texturizado virtual dinámico: Earthfly.

rio de Medio Ambiente y Medio Rural y Marino. La imagen utilizada tiene una resolución de 1048576 × 1048576 pixels (1 Terapixel) con tres canales de color de 8 bits (RGB888), lo cual resulta en 3 TBytes sin compresión y sin pirámides prefiltradas (*mipmaps*). La imagen está representada en proyección UTM huso 29N.

Información vectorial. Conjunto de datos vectoriales con información catastral de una zona urbana. Incluye primitivas de tipo línea, polilínea y textos. Esta información vectorial se representa sobre una textura virtual de 1048576 × 1048576 *texels*, con cuatro canales de color de 8 bits (RGBA8888). El motivo de usar cuatro canales es que resulta muy habitual mostrar la información vectorial combinada con una ortoimagen del terreno. Para ello se necesita el canal alfa a la hora de mezclar ambas texturas. Esta textura corresponde a una imagen de 1 Terapixel, o 4 TBytes de memoria considerando los cuatro canales de color. El espacio de la textura virtual utiliza la proyección UTM en huso 29N.

En el resto del capítulo se realizarán análisis de rendimiento en relación a diferentes aspectos del motor: *render* y actualización síncrona y asíncrona de texturas procedentes de datos *raster* y optimización y *render* a textura de los datos vectoriales. Finalmente se describirán algunas aplicaciones del sistema desarrollado, actualmente en producción en diferentes instalaciones.

5.1. *Render*

El primer análisis realizado corresponde al rendimiento del proceso de *render* del motor de texturas virtuales. Para ello, se toma como punto de partida una situación ideal en la que las cachés están completamente actualizadas y no se desplaza el centro de detalle ni se producen caducidades en la información. De esta forma se evitan las cargas de textura que afectarían al rendimiento del *render*, lo que se ha denominado en el capítulo anterior actualización síncrona.

La resolución de pantalla utilizada para todas las pruebas ha sido de 1280x1024 pixels. Se ha desactivado en el controlador de la tarjeta gráfica la sincronía entre el cambio de *buffer* y el barrido de la pantalla para permitir la generación de fotogramas a la mayor velocidad posible. Para centrar el análisis en el rendimiento del motor de texturizado, se ha minimizado la cantidad de geometría dibujada durante estas pruebas, limitándose a un polígono con toda la extensión de la textura mapeada sobre él.

Uno de los aspectos críticos tanto en la calidad como en el rendimiento del *render* es el tipo de filtrado utilizado, tal y como ya se ha descrito en capítulos anteriores. Por este motivo se han realizado las pruebas con las diferentes configuraciones posibles de filtrado de textura soportadas por el sistema.

En el caso de filtrado anisotrópico, la orientación del polígono respecto a la cámara afecta a la cantidad de muestras necesarias para aproximar el valor de cadal pixel. Para forzar una situación extrema, el polígono se encuentra en un ángulo cercano al de visión. En otras palabras, la vista es casi horizontal, considerando que el plano texturizado corresponde con el suelo (ver figuras J.1 a J.16).

Como referencia para contrastar el rendimiento alcanzado por la técnica desarrollada, se muestran las mediciones de tiempo para el polígono de prueba sin texturizar y el mismo polígono texturizado con una textura regular de OpenGL. Para igualar en la medida de lo posible ambas configuraciones, se ha utilizado como tamaño de textura en ambos casos el máximo soportado por el *hardware*: 8192×8192 *texels*.

5.1.1. Elección de los entornos de prueba

Para disponer de unos resultados lo más fiables y lo menos sesgados posible, las pruebas no se han limitado a una única configuración, sino que se han seleccionado diferentes configuraciones tanto *hardware* como *software*. La selección de máquinas se ha realizado dentro de los recursos accesibles por el autor, tanto personales como existentes en Videalab, puesto que no se dispuso de presupuesto alguno.

Los criterios de elección fueron orientados a cubrir la mayor variedad posible de situaciones, de forma que se puedan identificar desviaciones en los

parámetros estudiados debidos a configuraciones concretas. Dicho de otra manera, esto permite verificar si los comportamientos observados son coherentes a lo largo de las diferentes configuraciones de máquina y también analizar el impacto de los recursos *hardware* disponibles sobre diferentes aspectos del sistema.

Por este motivo, se han utilizado máquinas con prestaciones muy dispares y diferentes sistemas operativos (Windows XP, Windows Vista y Linux). La máquina más modesta fue un ordenador portátil de gama baja y la más potente una estación gráfica con procesador de cuatro núcleos y dos tarjetas NVIDIA GeForce 8800 Ultra en SLI. El detalle de las máquinas utilizadas en las pruebas está en la tabla 5.1.

En la tabla 5.2 se detallan las características de cada una de las pruebas realizadas en esta primera serie.

En el apéndice I se han incluido todas las mediciones realizadas en cada test. En esas gráficas (figuras I.1 a I.3) se puede observar que la configuración más potente (Shelob), con NVIDIA GeForce 8800 Ultra apenas ofrece mejora por utilizar dos tarjetas conectadas mediante SLI frente al uso de una sola tarjeta.

Por otra parte, la configuración menos potente (Bruinen), con una tarjeta NVIDIA GeForce M 8400 GS, muestra bastante inestabilidad en los datos, detectándose incluso unas oscilaciones periódicas que podrían estar provocadas por tareas del sistema. Aunque el sistema utilizado en esta configuración es Windows Vista, se desactivó el tema Aero que utiliza OpenGL para realizar ciertos efectos visuales sobre las ventanas y el escritorio, de forma que no consumiese recursos de *hardware* gráfico.

En las tablas I.1 a I.4 se resumen los valores medios de los datos obtenidos en las mediciones de tiempo de *render* de los tests 1 a 18 con las diferentes configuraciones *hardware*.

5.1.2. Comparación entre OpenGL y GeoTextura

Es evidente que los tiempos de *render* de GeoTextura no pueden mejorar los de las texturas regulares de OpenGL, pero resulta interesante hacer una comparación entre el rendimiento de ambos motores, para obtener una idea aproximada de la sobrecarga impuesta por la infraestructura de texturizado desarrollada. Se toma también como referencia el tiempo de *render* de la geometría de test sin ninguna textura aplicada. Este tiempo se puede considerar como una aproximación del tiempo empleado en el resto de las tareas a partir del cual se puede calcular una estimación del tiempo de texturizado.

En la tabla 5.3 se compara el rendimiento de ambas configuraciones de texturizado en igualdad de condiciones (filtrado por proximidad, misma resolución), junto con el *render* de la geometría sin aplicar textura de ningún tipo. Los valores mostrados en la tabla corresponden al promedio de fotogramas por segundo obtenidos en la ejecución de las pruebas.

Nombre	**Bruinen**
CPU	AMD Turion64 X2 1,90 GHz 2GB
GPU	NVIDIA GeForce 8400M GS 128MB
Sistema operativo	Windows Vista Home Premium 32-bit
Versión del controlador	179.48
Nombre	**Caradhras**
CPU	Intel Pentium D 3 GHz 4GB
GPU	NVIDIA GeForce 8800 GTS 320MB
Sistema operativo	Ubuntu Linux 32-bit
Versión del controlador	169.12
Nombre	**Shelob**
CPU	Intel Core2 Quad Q6600 2,4 GHz 3GB
GPU	NVIDIA GeForce 8800 Ultra 768MB
Sistema operativo	Windows XP Professional 32-bit
Versión del controlador	185.85

Cuadro 5.1: Configuraciones de *hardware* utilizadas para las pruebas de rendimiento del *render*.

Test	Motor de textura	Filtro	Núm. muestras
1	Sin textura	N/A	N/A
2	OpenGL	Proximidad	N/A
3	OpenGL	Bilineal	N/A
4	OpenGL	Trilineal	N/A
5	OpenGL	Anisotrópico	2
6	OpenGL	Anisotrópico	4
7	OpenGL	Anisotrópico	8
8	OpenGL	Anisotrópico	16
9	GeoTextura	Proximidad	N/A
10	GeoTextura	Bilineal	N/A
11	GeoTextura	Trilineal	N/A
12	GeoTextura	Anisotrópico	1
13	GeoTextura	Anisotrópico	2
14	GeoTextura	Anisotrópico	4
15	GeoTextura	Anisotrópico	8
16	GeoTextura	Anisotrópico	16
17	GeoTextura	Anisotrópico	32
18	GeoTextura	Anisotrópico	64

Cuadro 5.2: Primera batería de pruebas de rendimiento del *render*. Configuración de cada test.

	Bruinen	Caradhras	Shelob	Shelob (SLI)
Sin textura	225	4189	5910	5890
OpenGL	204	1745	5394	5393
GeoTextura	81	759	1454	1455

Cuadro 5.3: Comparación de rendimiento (fotogramas por segundo).

	Bruinen	Caradhras	Shelob	Shelob (SLI)
OpenGL	0.461	0.334	0.016	0.016
GeoTextura	7.865	1.078	0.518	0.517
Ratio	17.05x	3.23x	32.06x	33.07x

Cuadro 5.4: Estimación del tiempo de texturizado (ms) con OpenGL y GeoTextura.

Descontando el tiempo de *render* de la geometría (tiempo de *render* en el test 1), podemos estimar los tiempos de texturizado en los dos casos (OpenGL y GeoTextura) como los de la tabla 5.4, así como la ratio entre el tiempo de texturizado de GeoTextura y OpenGL. Resulta bastante destacable que la arquitectura *hardware* intermedia de las utilizadas en las pruebas es la que ofrece un mejor comportamiento del motor GeoTextura en comparación con las texturas de OpenGL. El *hardware* más potente es el que muestra la mayor pérdida de velocidad de *render* en la GeoTextura frente al texturizado estándar de OpenGL. Sin embargo, se observa que la diferencia está más en el tiempo de OpenGL que en el de GeoTextura. Esta última ofrece un comportamiento bastante coherente en las diferentes arquitecturas probadas, como se puede ver en la mitad derecha de la gráfica 5.2.

Como era de esperar, los resultados son claramente superiores en el caso de las texturas de OpenGL, por lo que para tamaños alcanzables por el máximo permitido en el *hardware* gráfico es recomendable utilizar ese tipo de texturas en la mayoría de los casos, aunque puede haber excepciones en las que interese utilizar GeoTextura por las posibilidades adicionales que ofrece frente a las texturas de OpenGL.

El motor de GeoTextura no sólo permite utilizar texturas de tamaño superior al límite del *hardware*, sino que también puede ser útil para manejar texturas más pequeñas con una porción de la memoria que necesitaría OpenGL para la textura completa, o simplemente mejorar la calidad visual de la textura tomando más muestras para el filtrado anisotrópico que las permitidas por el *hardware* gráfico (este límite se situaba en 16 muestras en el caso del *hardware* utilizado durante las pruebas de este trabajo).

En la figura 5.2 se comparan los tiempos de cada test (eje X) en las diferentes configuraciones *hardware*. Para visualizar mejor los datos, se han representado utilizando una escala logarítmica para el eje Y.

En las mediciones realizadas se ha incluido la configuración de GeoTex-

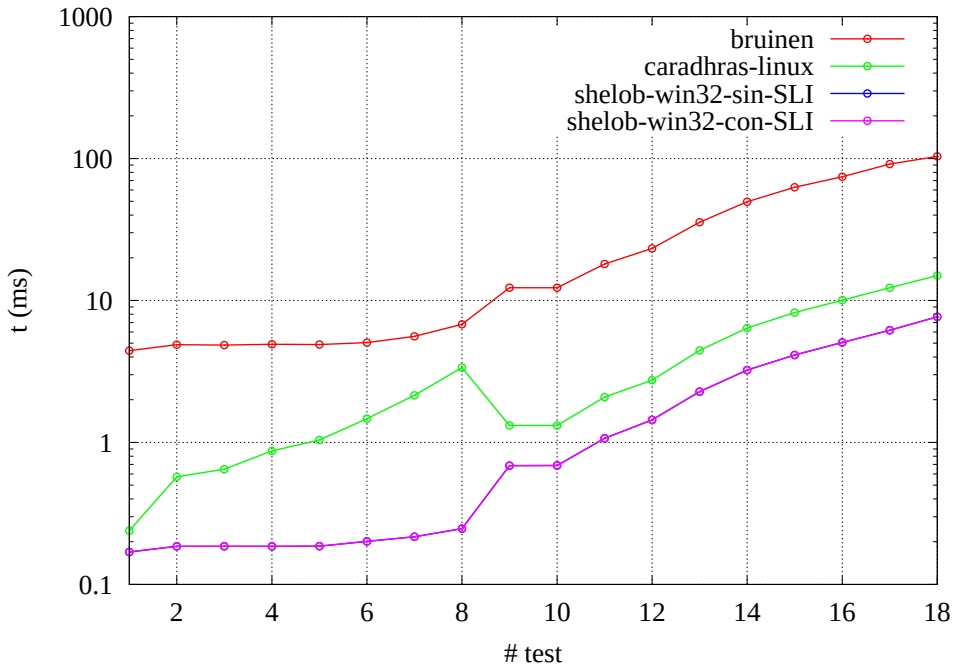


Figura 5.2: Tiempos de *render* por test en las diferentes configuraciones (ms, escala logarítmica).

	Bruinen	Caradhras	Shelob	Shelob (SLI)
Trilineal	18.069	2.085	1.070	1.070
Anisotrópico	23.280	2.744	1.441	1.441
Ratio	1.29x	1.32x	1.35x	1.35x

Cuadro 5.5: Comparación de tiempo de *render* (ms) entre filtrado trilineal y anisotrópico con una sola muestra.

tura con filtrado anisotrópico de una muestra. Aunque conceptualmente se trata del mismo caso que el filtrado trilineal, se ha incluido este test para medir la sobrecarga del código de filtrado anisotrópico respecto al trilineal. En la implementación realizada para estas pruebas, la sobrecarga impuesta por los cálculos adicionales del filtrado anisotrópico respecto al trilineal se muestran en la tabla 5.5, ascendiendo a un 35 % en el peor de los casos y a un 29 % en el mejor.

En las gráficas de la figura 5.3 se compara el tiempo de *render* de OpenGL (representado en rojo) con el del motor GeoTextura (representado en verde). En el eje X se muestran los diferentes tipos de filtrado disponibles en ambos sistemas (1: proximidad, 2: bilineal, 3: trilineal, 4: anisotrópico 2x, 5: anisotrópico 4x, 6: anisotrópico 8x, 7: anisotropico 16x). Las configuraciones de filtrado utilizadas en OpenGL son las que se describen en la tabla B.1. El

comportamiento del motor de GeoTextura es bastante similar en las diferentes arquitecturas *hardware*, mientras que el texturizado de OpenGL muestra una mayor variación en el rendimiento de los diferentes tipos de filtrado según la configuración *hardware*.

En las gráficas de la figura 5.4 se muestra el coste en tiempo de *render* por cada muestra anisotrópica, medido en milisegundos, comparando las implementaciones de OpenGL (rojo) y GeoTextura (verde). En las gráficas se puede apreciar que este tiempo es bastante constante en la implementación *hardware* de OpenGL. En la implementación de GeoTextura, basada en un *fragment shader*, tiene un coste considerablemente mayor. Sin embargo, se aprecia que hay una sobrecarga motivada por el código de este *fragment shader* que se amortiza según se incrementa el número de muestras.

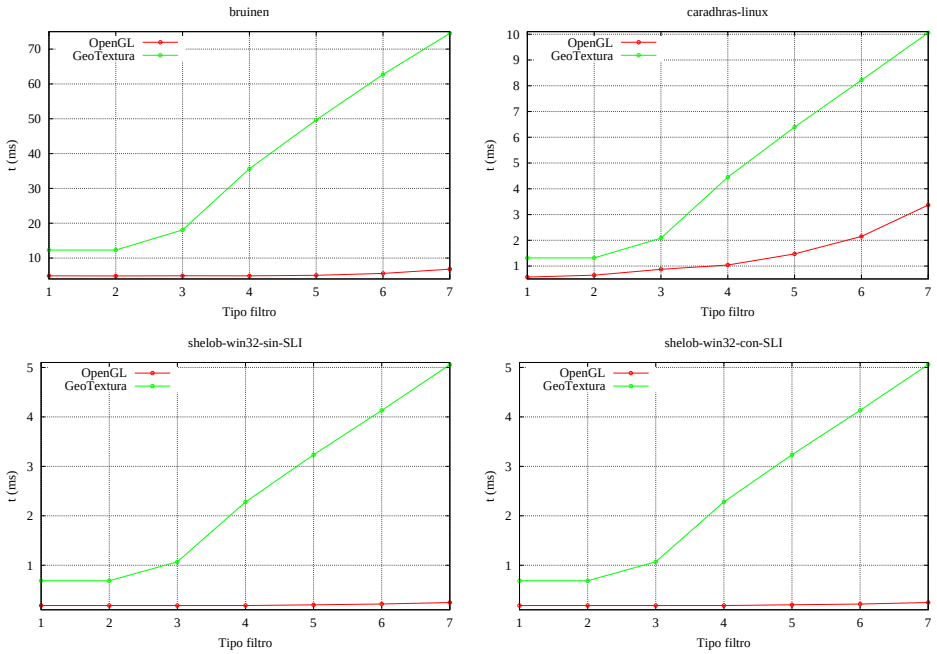


Figura 5.3: Comparación de tiempos de *render* con OpenGL y GeoTextura (Bruinen, Caradhras, Shelob y Shelob con SLI).

5.1.3. Impacto de la ventana cacheada en el rendimiento del *render*

En las pruebas realizadas en la primera serie (tests 1 a 18) y analizadas en la sección anterior se estableció el tamaño de la ventana cacheada en 2048×2048 *texels*. Se escogió este valor por ser el más adecuado para la resolución de pantalla utilizada (1280×1024 pixels), siempre que no haya restricciones a la memoria de vídeo disponible para la caché.

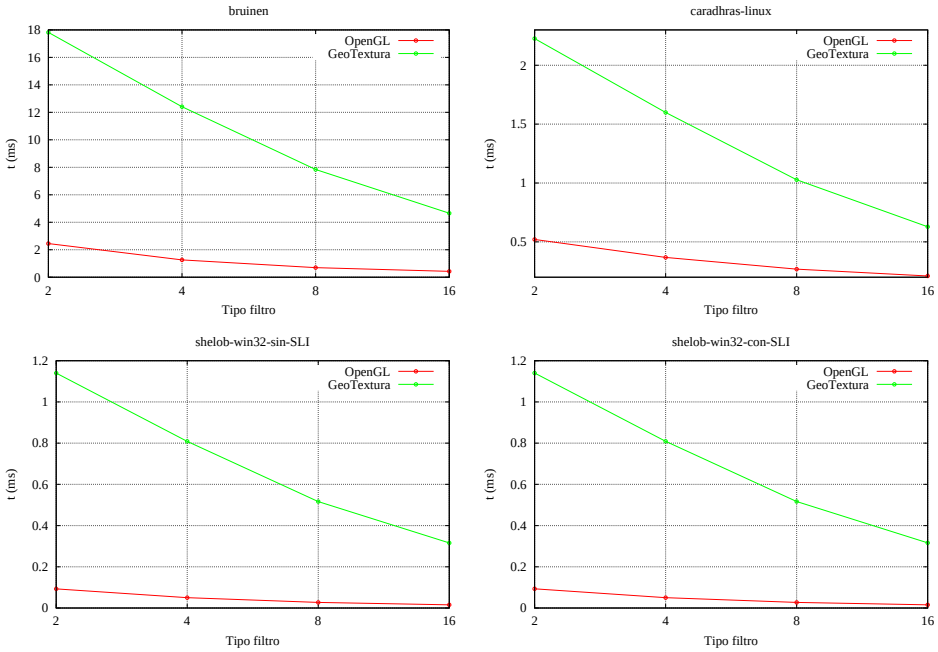


Figura 5.4: Tiempo de *render* por muestra anisotrópica (Bruinen, Caradhras, Shelob y Shelob con SLI).

En la siguiente serie de pruebas se analiza el impacto de este tamaño de ventana cacheada en el rendimiento del *render*. Se realizan las pruebas con el motor GeoTextura utilizando los mismos tipos de filtrado que en las pruebas de la primera serie para tamaños de ventana cacheada en TRAM desde 256 a 4096 *texels* de lado. Ambos extremos están fuera de la zona útil en la práctica, pero ayudan a ver muy claramente la tendencia del rendimiento en función del tamaño de ventana.

En las gráficas de la figura 5.5 se muestra cómo el tiempo de *render* se reduce con el uso de tamaños de ventana mayores. El motivo de que se mejore el rendimiento al incrementar el tamaño de la ventana cacheada en TRAM está en el funcionamiento del *fragment shader*, corazón del motor de texturizado, descrito en el apéndice B. Cuando la ventana cacheada es mayor, aumentan las probabilidades de que se disponga del nivel de textura necesario para un *fragment* y se necesiten menos iteraciones dentro del código del *fragment shader* para descender niveles buscando el de mayor detalle disponible para esa posición.

En el caso de la configuración *hardware* más modesta (Bruinen) se aprecia una subida en el mayor tamaño (4096) que contradice el comportamiento recién descrito. Esto está motivado porque ese tamaño de ventana con la configuración de GeoTextura empleada en estas pruebas necesita un tamaño de caché en TRAM de 144 MBytes, lo cual desborda la memoria de vídeo

de esa máquina.

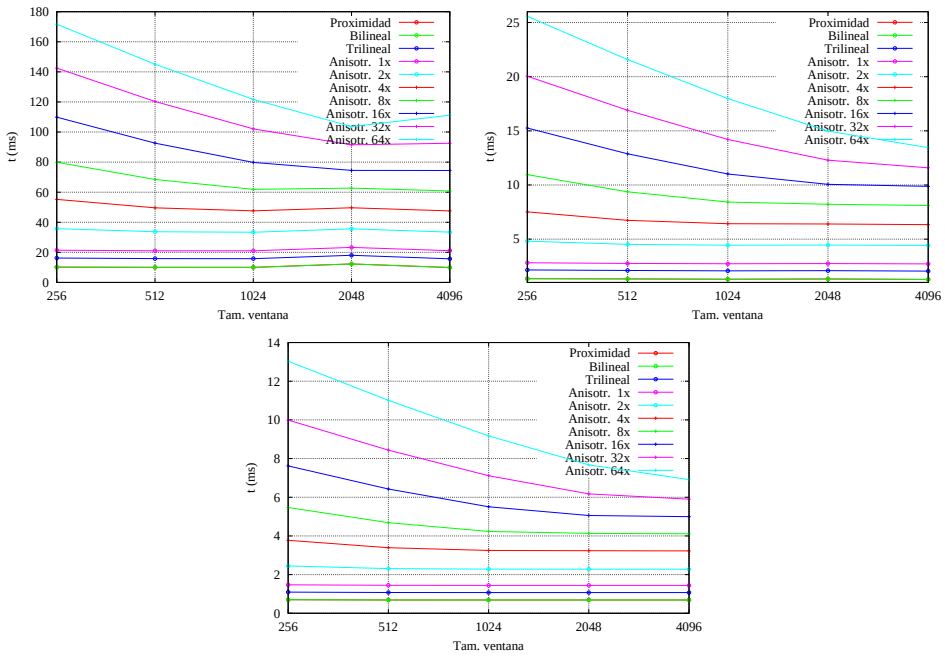


Figura 5.5: Tiempo de *render* por tamaño de ventana cacheada (Bruinen, Caradhras y Shelob).

5.1.4. Comparación de calidad entre OpenGL y GeoTextura

Para analizar la calidad del *render* obtenido en comparación con OpenGL se han tomado una serie de vistas desde la misma posición de cámara con ambos motores de texturizado.

Puesto que los cálculos para los filtros isotrópicos (proximidad, bilineal y trilineal) son bastante directos, el filtrado anisotrópico es el que a priori podría ofrecer más diferencias en la imagen final. Por este motivo se ha utilizado como textura una serie de imágenes con patrones reconocibles como cuadrículas o tableros de ajedrez.

Estas imágenes de test hacen muy notables los defectos del *render*, por lo que se podrían considerar como el peor caso posible. El mismo tipo de filtrado que muestra artefactos con estas imágenes de prueba, utilizado con una imagen de terreno, que por otra parte será el uso típico para este motor, ofrece una calidad excelente.

En las figuras J.1 a J.14 se muestran las parejas de *renders* realizados con ambos motores: OpenGL y GeoTextura. Calculando la resta de ambas imágenes se ha comprobado que existen algunas leves diferencias entre ellas, aunque estas no son especialmente notables a simple vista y no se puede concluir que ninguno de los dos sistemas sea mejor que el otro. Tal vez la

diferencia más importante es que la GeoTextura permite superar el límite de muestreo anisotrópico impuesto por el *hardware* (figuras J.15 y J.16), eso sí, limitándose a la zona de la textura cacheada en cada nivel.

5.1.5. Conclusiones

Tras los análisis realizados en cuanto al *render* de GeoTexturas, se pueden extraer las siguientes conclusiones:

- El rendimiento del *render*, aún siendo inferior al de las texturas de OpenGL, es perfectamente aceptable para una aplicación interactiva exigente.
- La calidad del texturizado es equivalente a la de OpenGL, con la ventaja adicional de que se puede incrementar el número de muestras para el filtrado anisotrópico por encima del límite impuesto por el hardware para las texturas de OpenGL.
- El incremento del tiempo en función del número de muestras anisotrópicas tiene un comportamiento aproximadamente lineal y bastante estable. En algunos casos, este comportamiento respecto al número de muestras mejora el de OpenGL (ver figura 5.2).
- El rendimiento del *render* de GeoTextura se ve afectado proporcionalmente al tamaño de la ventana cacheada.

5.2. Actualización síncrona

Tras el análisis del proceso de *render* realizado en la sección anterior, se continúa con el estudio del comportamiento de las cachés de los diferentes niveles existentes en la arquitectura desarrollada para GeoTextura. Se analiza el proceso de actualización de estas cachés, estudiando por separado la actualización síncrona de la caché de primer nivel (CacheTRAM) y la actualización asíncrona de la caché de segundo nivel (CacheRAM). En esta segunda, se tratará de manera independiente el caso de los datos de origen vectorial.

En esta sección se analizará el rendimiento de la actualización síncrona de la caché de primer nivel (CacheTRAM), que contiene en memoria de vídeo la información utilizada directamente en el *render*.

Para ello se centra el estudio en tres aspectos:

- El tiempo necesario para una recarga completa de la caché, y la tasa media de transferencia en este caso.
- El tiempo medio de carga de una tesela y la tasa de transferencia correspondiente.

- El impacto del tamaño o profundidad de pixel sobre la eficiencia de las transferencias.

Se analizan estos tres aspectos en las mismas configuraciones de máquina descritas en la sección de anterior y variando tanto el tamaño de ventana cacheada como el tamaño de tesela, para obtener un amplio abanico de resultados que cubra los posibles usos del sistema.

5.2.1. Recarga de la caché completa

Escogiendo el peor de los casos posibles, comenzaremos con una caché completamente vacía para medir el tiempo que necesita el sistema para cargar todos los datos para llenarla por completo en todos sus niveles de detalle. En este caso se realizan las pruebas con la base de datos de las imágenes de la Tierra (Blue Marble, enero 2004). Se realizan repetidas invalidaciones de la caché en diferentes zonas para forzar una recarga completa y se mide el tiempo consumido en esta operación.

Puesto que en estas pruebas consideramos únicamente la actualización síncrona, o lo que es lo mismo, la actualización del primer nivel de caché de textura (CacheTRAM), se realizan con toda la información necesaria disponible en la caché de segundo nivel (CacheRAM). De esta forma se asegura que las mediciones no se verán afectadas por el cuello de botella que se produciría en la carga desde los orígenes de los datos geoespaciales a memoria principal, es decir, la carga de la información en CacheRAM desde disco o desde la red. En otras palabras, se garantiza en estas pruebas que no se producirán fallos de caché en el segundo nivel (CacheRAM).

En las gráficas de la figura 5.6 se muestran los tiempos de recarga de la caché completa para las diferentes combinaciones de tamaño de ventana cacheada y tamaño de tesela para la actualización síncrona. Estos tiempos se han tomado como el promedio de un muestreo de 500 ciclos de actualización.

En esas gráficas se observa cómo el tiempo es obviamente mayor en los tamaños superiores de ventana, puesto que hay más cantidad de información a cargar. También se ilustra cómo afecta el tamaño de tesela de forma que con tamaños mayores se mejora la eficiencia, puesto que se amortiza más la sobrecarga de la actualización de cada tesela. La contrapartida es que al incrementar el tamaño de tesela se disminuye la granularidad en la ubicación de la ventana cacheada, lo cual, cuando el tamaño de tesela se acerca al tamaño de ventana puede producir cambios de detalle apreciables cuando se desplaza la ventana cacheada (los saltos serán del tamaño de tesela).

En las gráficas de la figura 5.7 se ha tenido en cuenta el tamaño de la caché, que se divide por el tiempo de la recarga mostrado en las gráficas anteriores para mostrar el nivel de eficiencia de cada configuración. Este nivel de eficiencia corresponde con la tasa de transferencia obtenida, medida en MBytes/s.

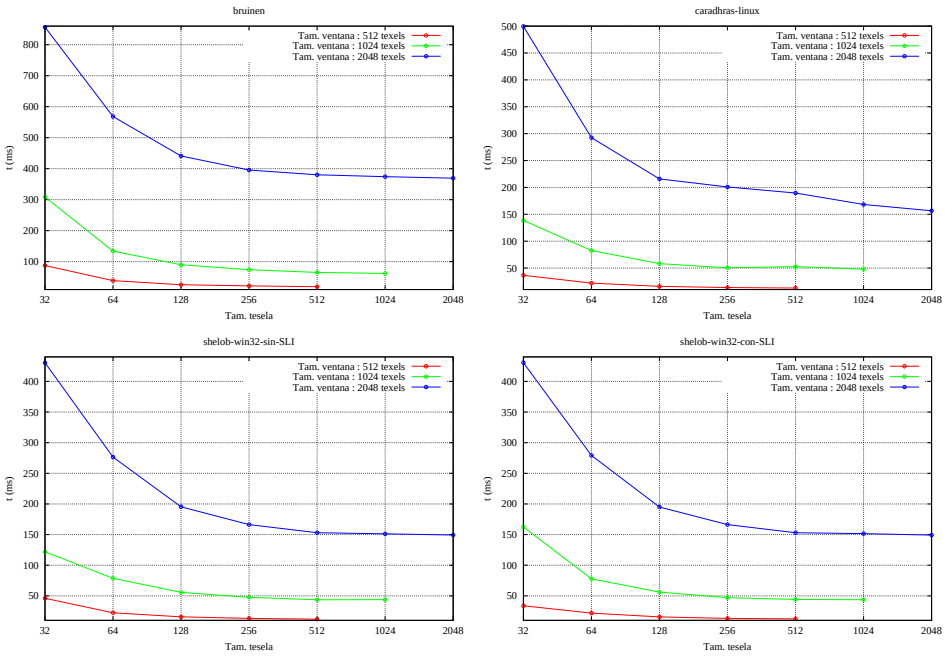


Figura 5.6: Tiempo de recarga de la caché completa (Bruinen, Caradhras, Shelob y Shelob con SLI).

En el caso concreto de la configuración de la máquina Bruinen (la más modesta de las configuraciones probadas) se observa un coste excesivo de la ventana de 2048×2048 *texels*, que no se produce en las otras configuraciones *hardware*. En la configuración intermedia (Caradhras) se observan algunas diferencias de rendimiento con los diferentes tamaños de ventana, especialmente en tamaños de tesela entre 128 y 512 *texels* de lado. La configuración más potente, aunque no ofrece un rendimiento excesivamente superior a la anterior, sí ofrece una mayor estabilidad, de forma que no se aprecian diferencias notables de rendimiento en la actualización entre los diferentes tamaños de ventana cacheada. Tampoco se observan diferencias importantes en este caso entre utilizar una única tarjeta o utilizar las dos tarjetas activando la opción SLI en el controlador.

El tiempo mostrado en las pruebas anteriores corresponde al ciclo completo de actualización de la textura, una parte del cual consiste en la actualización de los metadatos de la caché (realizada en la CPU) y otra parte consiste en la transferencia de las teselas requeridas desde RAM a TRAM. Estudiando la distribución de tiempos en estas operaciones, se ha comprobado que el tiempo consumido por las operaciones realizadas en CPU, básicamente actualización de metadatos, es al menos cuatro órdenes de magnitud inferior al tiempo total de actualización de la caché, por lo que se puede considerar despreciable a efectos prácticos. En concreto, estas operaciones, en las

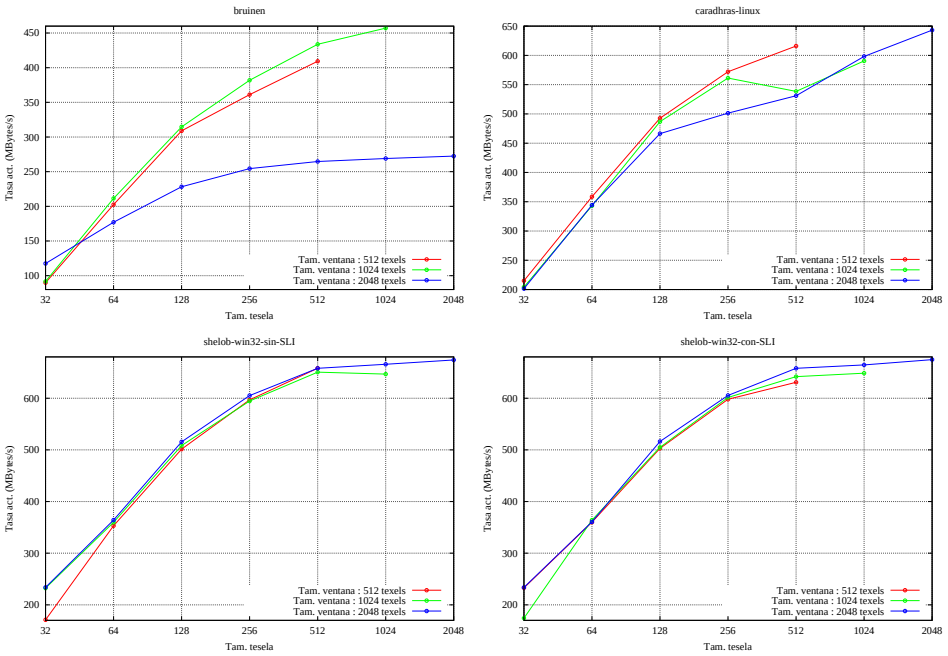


Figura 5.7: Tasa de transferencia de la recarga de la caché completa (Bruinen, Caradhras, Shelob y Shelob con SLI).

configuraciones utilizadas para las pruebas, suponen a lo sumo decenas de microsegundos.

5.2.2. Actualización de teselas

Para estudiar el tiempo dedicado a cargar cada tesela en TRAM, se han realizado mediciones del **tiempo de carga de cada tesela** sobre la misma configuración de las pruebas anteriores. Los resultados se toman del promedio de la actualización o carga de 50 000 teselas en TRAM. De nuevo se ha comprobado que el tiempo de actualización corresponde en la práctica únicamente al tiempo de carga de la tesela desde RAM a TRAM, el resto de operaciones que se realizan no suponen una sobrecarga apreciable.

Las gráficas de la figura 5.8 muestran los resultados de estas mediciones de tiempos de carga de teselas. Para apreciar mejor las diferencias, se incluyen a la izquierda las gráficas representadas en una escala lineal y a la derecha en una escala logarítmica. Todas las mediciones están representadas en milisegundos.

En las gráficas de la figura 5.9 se muestran las **tasas de actualización de las teselas**, medidas en MBytes por segundo, para las diferentes combinaciones de tamaño de ventana cacheada y tamaño de tesela estudiadas, que coinciden con las pruebas anteriores. Para apreciar mejor las diferencias, se incluyen a la izquierda las gráficas representadas en una escala lineal y a la

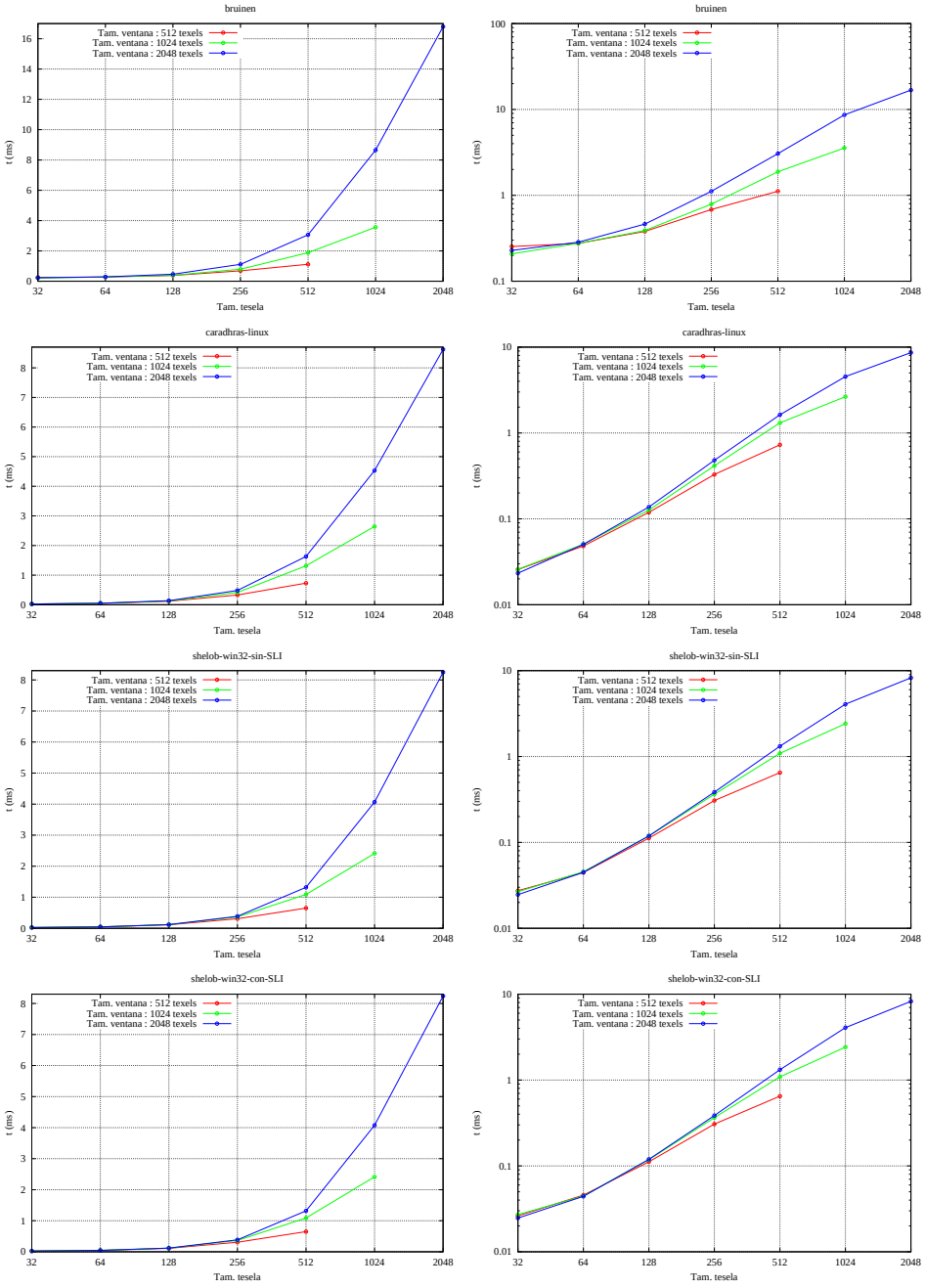


Figura 5.8: Tiempo medio de actualización de una tesela (Bruinen, Caradhras, Shelob y Shelob con SLI). Izquierda: escala lineal, derecha: escala logarítmica.

derecha en una escala logarítmica.

La eficiencia de las transferencias aumenta siempre con el tamaño de las teselas, lo cual es un resultado esperable. Pero también se puede apre-

ciar que esta tasa de transferencia se ve incrementada si la textura que se actualiza es de tamaño más próximo al de la tesela cargada. De hecho, la eficiencia se dispara de forma notable cuando coincide tamaño de tesela y tamaño de ventana, es decir, se actualiza la textura completa de una sola vez. Lamentablemente, esta situación no resulta adecuada para la técnica desarrollada, puesto que podrían producirse cambios de detalle visibles en primer plano, por lo que habrá que mantener el tamaño de tesela como máximo en la cuarta parte del tamaño de la ventana cacheada.

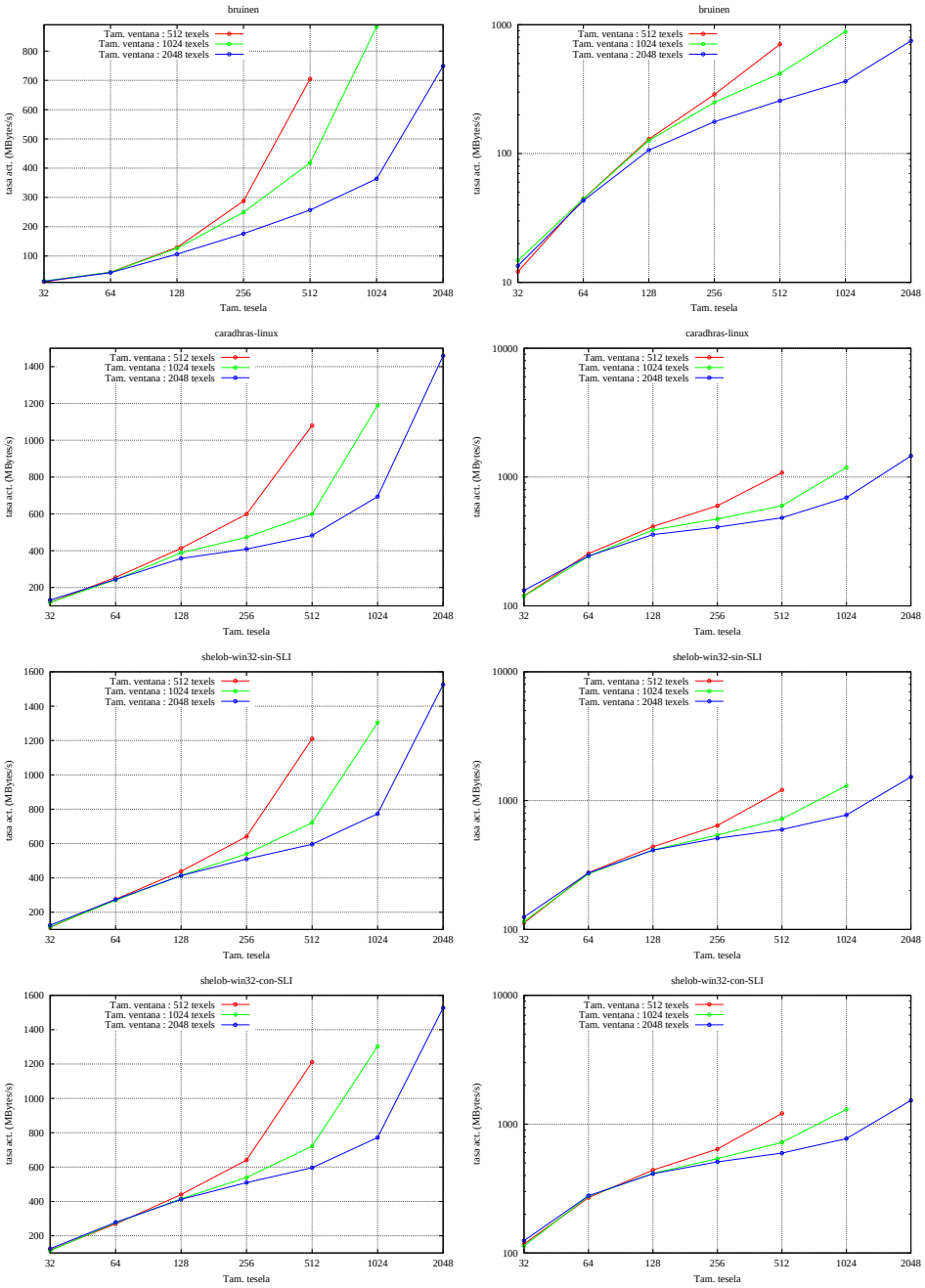


Figura 5.9: Tasa de actualización de una tesela (Bruinen, Caradhras, Shelob y Shelob con SLI). Izquierda: escala lineal, derecha: escala logarítmica.

Nombre	Canales	Tamaño	Descripción
INTENSITY16	1	16	Luminancia (escala de grises)
RGB888	3	24	RGB
RGBA8888	4	32	RGBα
BGRA8888	4	32	RGBα
RGBA5551	4	16	RGBα
RGB_S3TC_DXT1	3	4	RGB con compresión por HW

Cuadro 5.6: Formatos de pixel utilizados en las pruebas de transferencia.

5.2.3. Impacto del formato de pixel en el rendimiento de la actualización síncrona

Las mediciones de rendimiento mostradas para la actualización de la caché de textura en TRAM han sido realizadas utilizando un formato de pixel en color RGB con 8 bits por canal. Para estudiar el impacto del formato de pixel en la tasa de transferencia de las actualizaciones, se han realizado una serie de pruebas con una variedad de formatos de pixel con diferente tamaño y número de canales y en un caso utilizando un formato con compresión por *hardware*. Los formatos utilzados para estas pruebas se resumen en la tabla 5.6 (los tamaños se indican en bits por pixel).

Los valores representados en las gráficas se han calculado a partir del promedio de una muestra de 30 000 mediciones del tiempo de carga de teselas. Para facilitar el cálculo de la tasa de transferencia, se descartan las cargas de teselas inferiores al tamaño máximo (las correspondientes a los niveles inferiores de la pirámide).

En las gráficas de las figuras 5.10 a 5.12 se muestran las tasas de transferencia en cada configuración *hardware* y con tres tamaños de ventana cacheada: 512, 1024 y 2048 *texels* de lado, representadas en una escala lineal. Se representan las tasas (no los tiempos de transferencia) medidas en MBytes/s, por lo que se tiene en cuenta el tamaño en memoria del pixel. Nótese por tanto que con formatos de pixel de menor tamaño se podrán conseguir rendimientos mejores aunque la velocidad de transferencia sea peor. En la mayoría de estos casos, este incremento del rendimiento de la actualización irá en detrimento de la calidad, ya sea por dedicar menos bits a almacenar cada canal de la textura o bien por el uso de esquemas de compresión con pérdida.

En las gráficas se puede apreciar que el formato de pixel sí afecta a la tasa de transferencia, aunque el resultado es bastante dependiente del *hardware* utilizado. No se aprecia una tendencia clara en la mayoría de los formatos, los resultados son bastante variables, aunque sí se pueden extraer algunas conclusiones:

Uno de los aspectos interesantes a comentar sucede con los formatos RGBA8888 y BGRA8888 que aún conteniendo la misma información (si

bien ordenada de distinta forma) ofrecen diferencias notables de rendimiento y con un comportamiento perfectamente estable y coherente en todas las configuraciones *hardware* y en todos los tamaños de ventana y de tesela estudiados. La ordenación de los canales en BGRA ofrece un rendimiento siempre superior al RGBA, independientemente de los otros parámetros. Esto puede ser debido al funcionamiento interno del *hardware*, aunque es anecdótico que sea precisamente la forma más usada en la literatura (RGBA) la que experimenta un rendimiento más bajo.

Por otra parte, el formato más atípico, por su tamaño y por el hecho de usar la capacidad de compresión de la GPU, el S3TC DXT1 que almacena tres canales de color (RGB), muestra unos resultados muy variables respecto a los otros formatos de pixel estudiados. De hecho, en la configuración más modesta (Bruinen) y en la más potente (Shelob) ofrece unos resultados muy pobres, tanto en términos absolutos (del orden de 100 y 200 MBytes/s respectivamente) como en relación a los otros formatos. En cambio, en la configuración intermedia (Caradhras) muestra un rendimiento próximo a los otros formatos o incluso superior en algunos casos.

En general, el formato más utilizado, color RGB de 24 bits (RGB888), presenta el rendimiento más bajo de los formatos estudiados sin considerar el formato comprimido (S3TC DXT1).

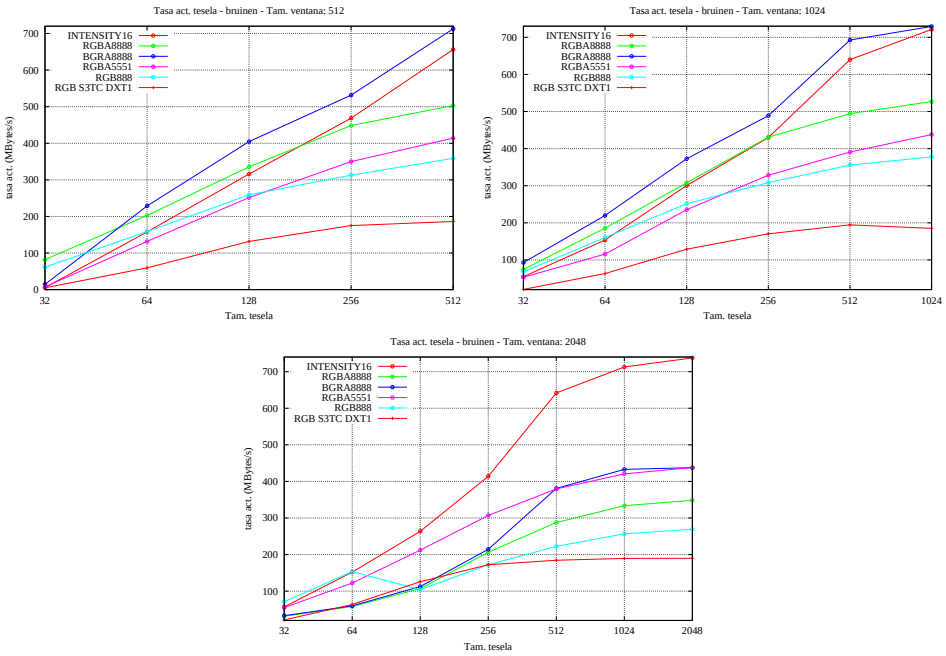


Figura 5.10: Tasa de actualización de una tesela según formato de pixel (Bruinen). Tamaños de ventana: 512, 1024 y 2048 *texels*.

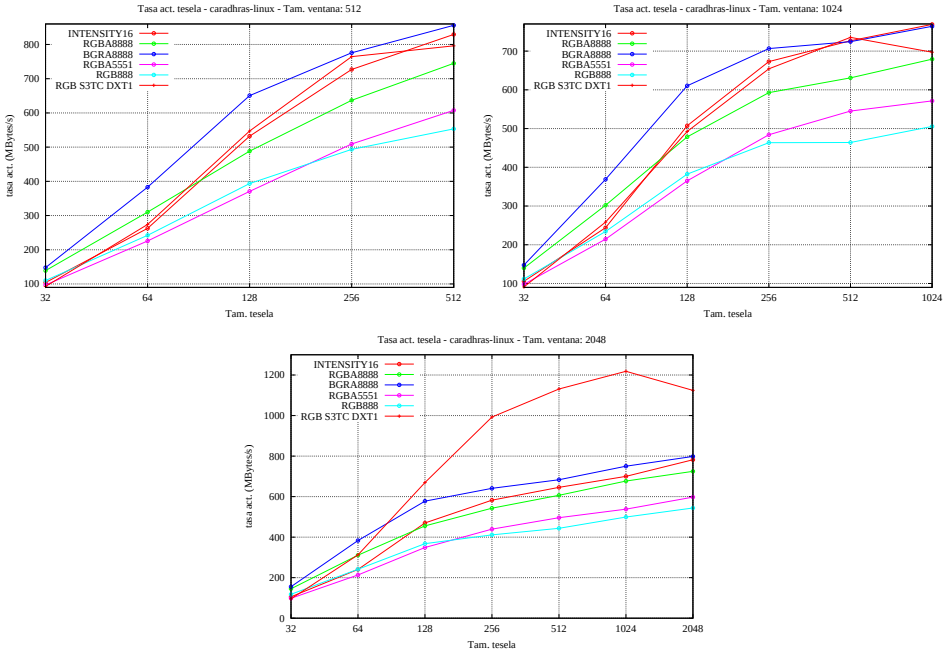


Figura 5.11: Tasa de actualización de una tesela según formato de pixel (Caradhras). Tamaño de ventana: 512, 1024 y 2048 *texels*.

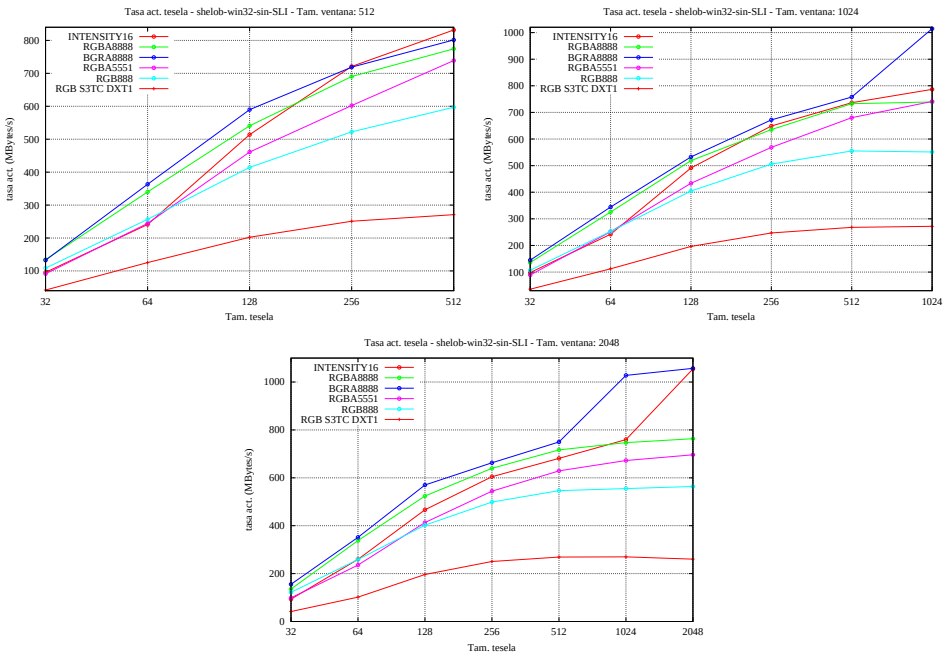


Figura 5.12: Tasa de actualización de una tesela según formato de pixel (Shelob). Tamaño de ventana: 512, 1024 y 2048 *texels*.

5.2.4. Conclusiones

De las pruebas realizadas en cuanto a la actualización síncrona de GeoTextura, se pueden extraer algunas conclusiones, que se describen a continuación:

- El tiempo de recarga completa de la caché es más que aceptable para una aplicación interactiva. Este tiempo es siempre inferior a un segundo y en la configuración habitual oscila entre 150 y 400 ms.
- El tamaño de tesela óptimo está situado en torno a los 256 o 512 *texels* de lado. Estos tamaños ofrecen una granularidad adecuada para la distribución de la actualización en varios ciclos y no sufren una penalización grave a la eficiencia de las transferencias.
- El tiempo medio de recarga por tesela, en todas las configuraciones, está en torno al milisegundo para los tamaños de tesela habituales (entre 256 y 512 *texels* de lado), por lo que la recarga de la caché se puede distribuir muy bien en diferentes ciclos de render, de forma que no se afecte al *frame rate* de la aplicación.
- El tamaño de ventana afecta de forma notable a la tasa de transferencia de las teselas. A mayor tamaño de ventana, menor tasa de transferencia.
- El formato de pixel BGRA8888 ofrece en promedio la mejor tasa de transferencia en todas las configuraciones, y supera siempre al formato RGBA8888, que contiene la misma información en diferente orden.
- El formato comprimido S3TC DXT1, si bien no ofrece un rendimiento especialmente bueno en cuanto a la tasa de transferencia en comparación con los otros formatos, es una opción muy interesante por el alto grado de compresión (4 bits por pixel). Esto redunda en un triple beneficio: reduce el almacenamiento en el origen de datos, reduce las necesidades de transferencia por el bus entre memoria principal y memoria de vídeo y reduce la ocupación de memoria de vídeo. Todas estas reducciones son a la sexta parte de lo que se necesitaría para almacenar esos tres canales en formato RGB888, formato que por otra parte ha demostrado ser el menos eficiente de los estudiados en cuanto a tasa de transferencia.

El hecho de haber estudiado el comportamiento de la caché en configuraciones de máquina tan dispares y variando los parámetros de tamaño de ventana cacheada, tamaño de tesela y formato de pixel, permite extrapolar los resultados para:

- Elegir la configuración hardware necesaria para una aplicación concreta del motor GeoTextura.

- A partir de una configuración *hardware* disponible, determinar la configuración óptima del motor GeoTextura que combine calidad, rendimiento y necesidades de almacenamiento. Es decir, ayuda a tomar la decisión crítica de establecer los parámetros de tamaño de ventana para la caché, tamaño de tesela y formato de pixel.

Esto ha resultado de gran utilidad en los proyectos reales donde se ha aplicado el motor GeoTextura, para determinar la configuración de los ejemplos de aplicación que se detallan en la sección 5.5.

5.3. Actualización asíncrona

Una vez analizado el rendimiento de los procesos de actualización síncrona, tal y como se ha descrito en el apartado anterior, nos centramos en el estudio del mecanismo de actualización asíncrona, es decir, la actualización de la caché de segundo nivel (CacheRAM). En este caso se trata de mantener en la memoria RAM o memoria principal del ordenador la información que va a ser solicitada por la cache de primer nivel (CacheTRAM).

En este trabajo se ha diseñado e implementado una caché predictiva (denominada **CacheRAMPrecarga**), que ya ha sido descrita en la sección 4.5, y cuyo rendimiento será comparado con otra implementación de CacheRAM más sencilla (denominada **CacheRAMLRUPrio**), desarrollada con anterioridad a esta tesis.

CacheRAMLRUPrio tiene un comportamiento reactivo en cuanto a la carga de los bloques solicitados, frente al comportamiento predictivo de CacheRAMPrecarga. Por este motivo, en este texto se hará referencia a ambas cachés como **predictiva** y **reactiva**.

En esta sección se analiza en detalle el comportamiento de la caché predictiva en diferentes situaciones de uso y se contrasta con el de la caché reactiva. A continuación se describe brevemente el funcionamiento de esta caché reactiva, para detallar luego las pruebas realizadas y analizar sus resultados.

5.3.1. Descripción de la caché de referencia

El comportamiento de la caché reactiva, se puede resumir de la siguiente manera: cuando se solicita un *buffer* a la caché, si no estaba presente, se crea una petición de carga, que será potencialmente atendida por algún *thread* de carga asíncrona. Estas peticiones se almacenan en una estructura que guarda únicamente la última petición de cada nivel, y el *thread* de carga las prioriza por orden de nivel, de menor a mayor detalle.

El motivo de no acumular peticiones es que en situaciones de carga alta, el tiempo de validez de dichas peticiones puede ser muy limitado, por lo que no tiene sentido atender más que la más reciente. El motivo de dar más

prioridad a los detalles inferiores es porque cubren un área mayor de terreno y por lo tanto, tendrá un tiempo de validez superior.

Se mantiene el número de *buffers* reservados en RAM para la caché y cuando hayan sido utilizados todos y haga falta cargar un bloque nuevo, se reutilizará alguno de los *buffers* existentes, siguiendo una política LRU (se toma el *buffer* menos recientemente solicitado a la caché).

5.3.2. Prueba 1. Disco local, sin compresión.

La primera configuración de pruebas utiliza la aplicación Earthfly con la imagen de Blue Marble correspondiente a enero de 2004 mapeada sobre una geometría esférica que emula (a gran escala) la forma del planeta Tierra. Los datos de dicha imagen se obtuvieron de un disco duro externo con interfaz USB 2.0, de forma que el rendimiento de los accesos a disco debería ser similar en las diferentes configuraciones *hardware* utilizadas. La tasa de transferencia observada en las pruebas está en torno a los 20 MBytes/s. Con objeto de intentar minimizar las variaciones (debido a parámetros externos a los medidos) en el rendimiento de la caché entre las diferentes pruebas, el cargador de *buffers* no hace uso del sistema de "*buffer cache*" del sistema operativo. El uso del "*buffer cache*" del sistema operativo puede mejorar el rendimiento en algunos casos, pero el comportamiento no es predecible, y en otros casos puede suponer una sobrecarga inútil, puesto que se están cacheando los mismos datos dos veces en memoria RAM.

Para reproducir el mismo comportamiento en las diferentes pruebas el movimiento está controlado por una ruta automática. La trayectoria de la cámara sigue una función sinusoidal en latitud y se desplaza a velocidad lineal constante en longitud. Esto supone uno de los peores casos para la caché en RAM, puesto que no se vuelve a visitar la misma zona hasta dar una vuelta completa. Podemos asumir por lo tanto, que el comportamiento en un uso normal del motor de texturizado sería no inferior al mostrado en estas pruebas, y con gran probabilidad notablemente superior.

Las pruebas se han realizado para tamaños de *buffer* de 512, 1024 y 2048 *texels* de lado, 768 kBytes, 3 MBytes y 12 MBytes respectivamente para el formato de pixel utilizado: RGB888 (24 bits por pixel). Se han utilizado en estas pruebas cuatro velocidades de vuelo diferentes para comprobar su impacto en el rendimiento de la caché. En la tabla 5.7 se muestran la velocidad media en cada caso, representada en grados por segundo, nudos, kilómetros por hora y número *Mach*.

Los parámetros que se han analizado para la combinación de opciones de configuración descritas han sido los siguientes:

Solicitud de *buffers* Tiempo de ejecución de la llamada de solicitud de un *buffer*. Esta parte es crítica, puesto que se ejecuta en el *thread* de *render*, durante la actualización síncrona. No se realizan en ella las

Vel	º/s	Nudos	km/h	*Mach*
1	0,12	27 367	50 684	4.1
2	1,26	273 672	506 840	41
3	12,67	2 736 720	5 068 400	413
4	126,70	27 367 200	50 684 000	4 137

Cuadro 5.7: Velocidades de vuelo de las pruebas de caché.

cargas desde disco o desde la red, sólo se devolverá el *buffer* si ya está disponible en la caché en RAM.

Aciertos y fallos de caché Resultado de las peticiones a la caché en RAM a partir de la solicitud de *buffers*. Se registran tanto los aciertos como los fallos de caché producidos durante la ejecución de las pruebas. Estos valores son la clave para medir la eficiencia de la caché predictiva, puesto que su fin último es minimizar los fallos de caché.

Actualización de la caché en TRAM Porcentaje de actualización de la caché en TRAM (*texels* cargados frente a *texels* totales de la caché) en cada fotograma. El objetivo es mantener este valor en el 100 %, para lo cual se deben reducir a 0 los fallos de caché. Es importante destacar que el objetivo de la caché es minimizar los fallos, no mantener esta actualización en un valor alto. Existen otros parámetros que influyen en el nivel de actualización de la caché en TRAM, además de la eficiencia de la caché en RAM, como el orden de solicitud de las teselas en cuanto a los niveles de detalle de la textura. Este aspecto se explica en detalle más adelante en este capítulo.

***Thread* de cargas asíncronas** Intervalos de tiempo de actividad y de espera del *thread* de cargas asíncronas. No se conoce realmente el uso efectivo de CPU, estas mediciones indican únicamente cuáles han sido los intervalos de tiempo con el proceso de cargas activo (incluyendo la propia carga, que no consume CPU) y en cuáles ha estado el *thread* bloqueado a la espera de nuevas peticiones.

***Thread* de precarga** En el caso de la caché predictiva, se miden los intervalos de espera y actividad, al igual que en el *thread* de cargas asíncronas, para el *thread* de precarga, encargado de generar las listas de *buffers* a cargar en el *thread* de cargas asíncronas.

Las mediciones realizadas corresponden a los promedios de esos parámetros en una ejecución durante 500 000 peticiones o 5 000 cuadros, lo que sucediese primero.

Desbordamientos del tiempo de cuadro

Durante las pruebas realizadas se ha detectado algún desbordamiento esporádico del tiempo de cuadro. No se ha observado ningún patrón reconocible asociado al tamaño de bloque o al tipo de caché. Es en las velocidades de vuelo más elevadas donde se produce mayor número de desbordamientos, pero esto sólo resulta grave en el caso de la máquina más modesta, en la que por otra parte, ya se producen sin utilizar el motor de texturizado, por lo que las causas de dichas pérdidas quedan fuera del alcance de este trabajo. Estos desbordamientos se pueden controlar con el subsistema de control y gestión de carga, descrito en el apéndice H. Para ello se ajustará el margen de seguridad establecido para evitar desbordamientos de tiempo de cuadro en función de la configuración utilizada. Para las pruebas analizadas en este capítulo se utilizó un margen de seguridad de 0,5 ms.

Tiempo de solicitud de *buffers*

Respecto a los tiempos de petición de *buffers*, las mediciones indican que podemos despreciarlos, puesto que son insignificantes en relación al tiempo de un cuadro. En concreto, en todos los casos probados está en el orden de microsegundos (todos los promedios de tiempo son inferiores a 7 μs).

Rendimiento de la caché: ratio de aciertos

Las gráficas de la figura 5.13 ilustran el comportamiento de ambas cachés (reactiva y predictiva) en las tres máquinas de prueba (Bruinen, Caradhras y Shelob), comparando las cuatro velocidades de la tabla 5.7 y los tres tamaños de bloque: 512, 1024 y 2048. El valor representado en las gráficas es el promedio de las ratios de aciertos de caché (i.e. el cociente entre el número de aciertos y el número total de peticiones). Este valor es el que mide realmente la eficiencia de la caché, puesto que el objetivo de dicha caché es maximizar los aciertos o lo que es lo mismo, minimizar los fallos.

La primera gráfica de cada pareja muestra las ratios representadas en función del **tamaño de *buffer***. Las velocidades de vuelo se codifican en los colores de las líneas (1:rojo, 2:verde, 3:azul, 4:magenta). Los valores de la caché reactiva se representan con círculos, los de la caché predictiva con aspas.

La segunda gráfica de cada pareja muestra las ratios en función de la **velocidad de vuelo**. Los tamaños de *buffer* se codifican en los colores de las líneas (512:rojo, 1024:verde, 2048:azul). Los valores de la caché reactiva se representan con círculos, los de la caché predictiva con aspas.

En dichas gráficas se puede ver cómo la caché predictiva cumple su objetivo de minimizar los fallos de caché, llegando a reducirlos a cero en velocidades moderadas. Es en estas velocidades (1 y 2) cuando la caché se muestra

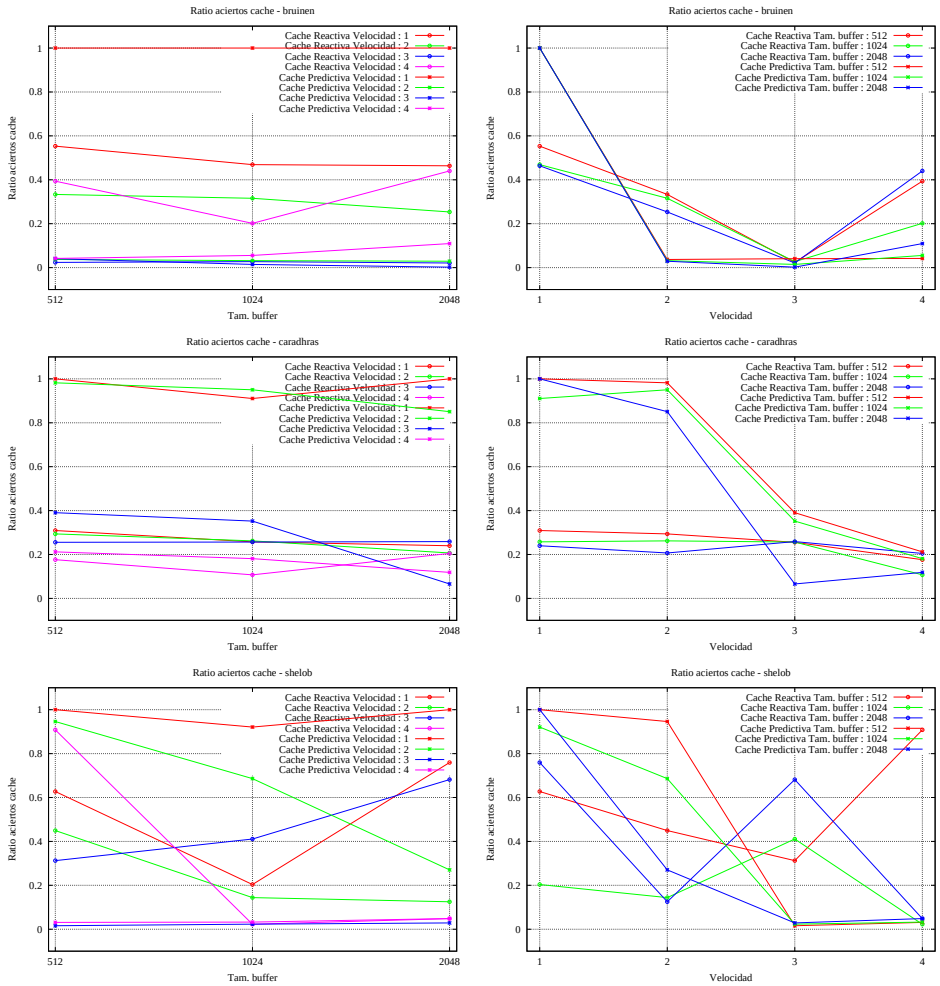


Figura 5.13: Ratio de aciertos de caché (Bruinen, Caradhras y Shelob). Izquierda: por tamaño de *buffer*, derecha: por velocidad.

claramente superior independientemente de la configuración *hardware* y el sistema operativo utilizado.

En velocidades más altas, la caché predictiva pierde su ventaja, puesto que el tiempo de validez de los *buffers* cargados se reduce hasta hacer inútil la predicción. En estos casos el comportamiento es similar al de la caché reactiva, y deficiente en ambos casos, puesto que en esa situación la elevada velocidad de vuelo hace inútil la caché en RAM, sea cual sea la técnica utilizada. También es cierto que en estos casos de velocidades altas, el usuario no puede apreciar el detalle del terreno, por lo que la caída del mismo no es un problema grave. El uso habitual del sistema, y donde se demanda una alta calidad en la visualización, es con movimientos lentos y suaves o incluso con la cámara detenida para poder observar con atención la zona del terreno

que interese al usuario.

Es importante mencionar que las pruebas se han hecho en el peor caso posible, con un movimento contínuo de la cámara, sin volver a pasar por la misma zona. La caché predictiva ofrecerá mayor ventaja cuando el usuario se detenga en ocasiones a observar diferentes zonas de interés, puesto que este tiempo es aprovechado para precargar las zonas visitables en un futuro. Este comportamiento, por otra parte, será el habitual en las aplicaciones de visualización de información geográfica, por lo que podemos considerar los resultados aquí expuestos como una cota inferior del rendimiento esperable.

Impacto del tamaño de *buffer* en el rendimiento de la caché

Respecto a los tamaños de bloque utilizados en las pruebas, los resultados son bastante variables. En general, el tamaño de bloque superior no ofrece una ventaja significativa en cuanto a rendimiento de las transferencias y sí produce una cierta inestabilidad en los resultados, que en este extremo se muestran más variables.

El tamaño de bloque más grande de los utilizados en las pruebas (2048) ofrece una muy ligera mejora en el ratio de aciertos de caché en la caché reactiva, pero una caída significativa en la predictiva. Esto se ha observado sobre todo en la configuración más potente (Shelob). Este tamaño de bloque sin embargo, tiene importantes desventajas:

- Los requerimientos de memoria reservada para la caché se disparan para cubrir la misma zona en de la caché en TRAM. Para los niveles incompletos de la textura (los niveles de la pila) es necesario disponer de al menos cuatro *buffers* para cubrirlos (zona A).

- La granularidad de las cargas es excesivamente gruesa. En un sistema tan dinámico como este, donde el tiempo de validez de la información puede ser muy efímero, el hecho de cargar bloques muy grandes, puede provocar que cuando finalice la carga la información ya no sea válida por haber quedado fuera de la ventana cacheada en TRAM. Por este motivo, es preferible realizar varias cargas pequeñas en lugar de una grande, aunque la velocidad de transferencia sea un poco menor.

Una desventaja de los tamaños de bloque pequeños es que se dispara exponencialmente el número de ficheros y directorios en disco, lo cual incrementa la sobrecarga del manejo de dichos ficheros en disco. Sin embargo, esta desventaja afecta principalmente a las operaciones de mantenimiento de las colecciones de datos (creación, borrado, copia, cambio de permisos, propietarios, etc), pero en la ejecución no será apreciable, puesto que el tiempo de acceso a ficheros y directorios es despreciable en comparación al tiempo de las cargas.

Observando comparativamente las tres configuraciones, se aprecia que es en la más modesta (Bruinen) donde existe una cierta ventaja de la caché reactiva frente a la predictiva.

La caché predictiva tiene ventajas evidentes y cumple su objetivo de minimizar los fallos de caché, sin embargo también se hace notable que tiene un coste más elevado, por lo que su potencial se aprovecha al máximo cuando los recursos de *hardware* son mayores.

Porcentaje de actualización

Estudiando el nivel de actualización de la caché en TRAM, el tamaño de *buffer* no afecta significativamente al nivel de actualización de dicha caché, como se puede ver en la figura 5.14.

En la máquina más modesta (Bruinen), la caché reactiva ofrece un comportamiento notablemente mejor (en cuanto al nivel de actualización de la caché en TRAM) que la predictiva, especialmente en las velocidades altas. Esto es debido a los limitados recursos de esa máquina, insuficientes para la calidad exigida en la configuración de prueba más allá de la velocidad 1.

En la máquina más potente de las tres configuraciones (Shelob) es donde se aprecia una clara ventaja de la caché predictiva frente a la reactiva, incluso en las velocidades altas. Al disponer de cuatro núcleos, se paralelizan mejor las tareas y esto se ve reflejado en el rendimiento de la caché.

En la configuración intermedia, basada en Linux, el rendimiento de la caché predictiva es inferior en las velocidades altas y superior en las velocidades bajas, aunque a estas velocidades ambas cachés ofrecen buenos resultados, mejores incluso que los obtenidos en la configuración más potente. Esto puede ser debido a la diferente gestión de los *threads* y/o de los accesos a disco en Windows XP y Linux.

El tamaño de *buffer* tiene una influencia pequeña en el nivel de actualización en TRAM en todas las configuraciones probadas, existen ligeras diferencias en algunos casos, pero sin llegar a ofrecer una tendencia clara que permita sacar conclusiones.

Diferencia cualitativa: orden de carga de los *buffers*

Observando los resultados del nivel de compleción de la caché en TRAM, se puede apreciar que no se corresponde la calidad de la caché predictiva en cuanto a fallos de caché (figura 5.13) con la actualización en TRAM (figura 5.14). Esto es debido a una diferencia entre las dos cachés, comentada anteriormente: la forma de determinar el orden de prioridad de los niveles de detalle de la textura es distinto. La caché reactiva aplica un orden de carga de abajo hacia arriba (de menor a mayor detalle), mientras que la caché predictiva determina el nivel principal observado en pantalla a partir

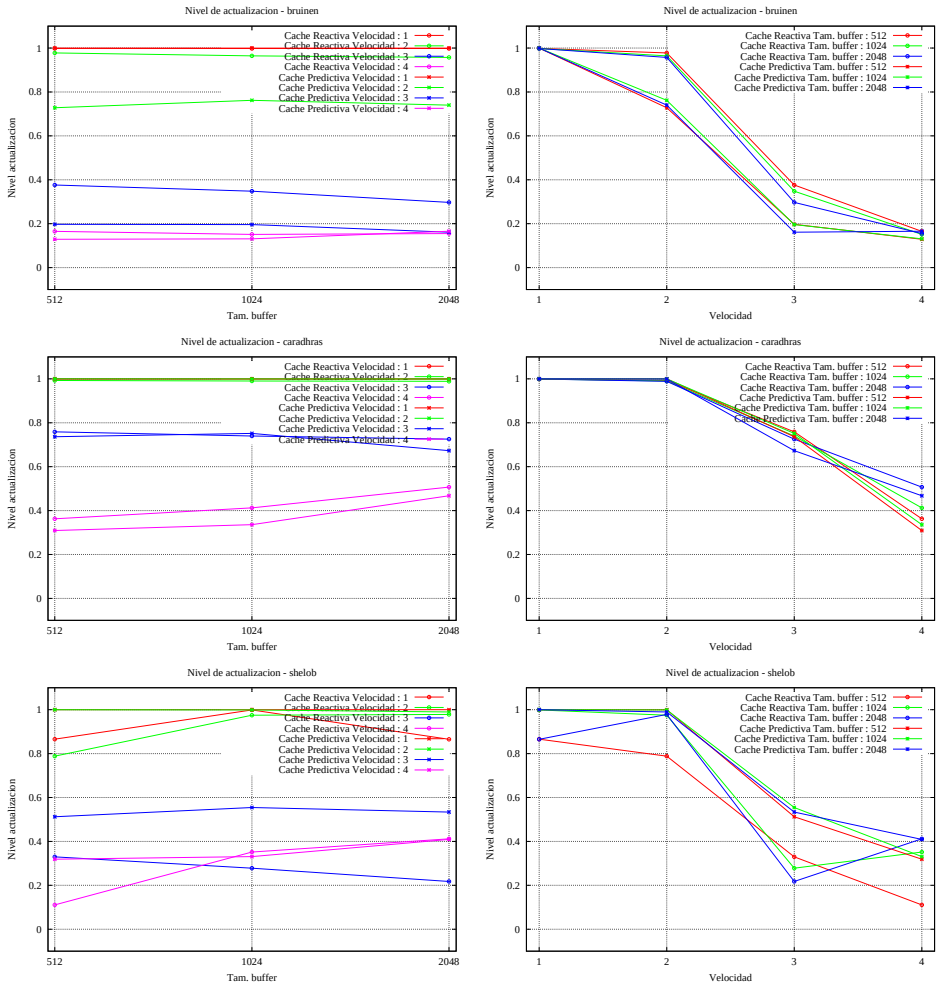


Figura 5.14: Nivel de actualización de la caché en TRAM (Briunen, Caradhras y Shelob). Izquierda: por tamaño de *buffer*, derecha: por velocidad.

de la posición de la cámara, y realiza una carga centrada en ese nivel. Este comportamiento ha sido descrito en la sección 4.4.1.

Esta diferencia en el orden de carga produce dos efectos:

- Los niveles cargados por la caché reactiva tendrán un mayor tiempo de vida, puesto que a menor detalle, mayor extensión geográfica. De ahí que los resultados numéricos en nivel de compleción de la caché se vean incrementados.

- La calidad observada visualmente será superior en el caso de la caché predictiva, puesto que los niveles cargados serán los que se están mostrando en primer plano, aunque para ello se sacrifiquen niveles inferiores (con mayor esperanza de vida) y por tanto en la medida de porcentaje

Tam. *buffer*	Velocidad	Bruinen	Caradhras	Shelob
512	1	0,12	0,01	0,03
512	2	0,88	0,08	0,24
512	3	5,20	1,46	3,20
512	4	6,95	4,55	7,60
1024	1	0,10	0,01	0,02
1024	2	0,51	0,05	0,16
1024	3	3,06	1,00	2,25
1024	4	3,70	1,65	2,72
2048	1	0,06	0,01	0,02
2048	2	0,35	0,05	0,15
2048	3	1,90	0,47	1,17
2048	4	2,50	0,67	1,83

Cuadro 5.8: Porcentajes de tiempo de actividad de los *threads* de precarga.

de *texels* presentes en la caché el resultado numérico sea peor.

Por tanto, el orden de carga de los niveles, aunque empeore los resultados numéricos (en cuanto a porcentaje de compleción de la caché en TRAM) de la caché predictiva, provoca que visualmente los resultados sean mejores.

Actividad de los *threads* de confección de listas de carga

Respecto a los tiempos de actividad de los *threads* auxiliares para las cargas asíncronas, el *thread* encargado de la generación de listas de *buffers* a cargar/precargar de la caché predictiva ha demostrado ser muy ligero. Su tiempo de actividad es inferior al 10 % en todos los casos y sólo en las velocidades más altas y el tamaño de *buffer* menor (y que por tanto produce mayor cantidad de *buffers*) supera el 5 %. En la tabla 5.8 se resumen estos porcentajes de tiempos de actividad. Como resultado interesante, se aprecia que la configuración de Linux (Caradhras) reduce el tiempo de actividad del *thread* de precarga respecto a las de Windows, a pesar de disponer de menos núcleos que la CPU de Shelob (dos frente a cuatro).

Actividad de los *threads* de carga

Los *threads* encargados de realizar las cargas asíncronas sí están la mayor parte del tiempo en actividad, puesto que se incluye el tiempo de carga dentro de esa actividad. Sin embargo, esta actividad es mayormente de entrada salida, por lo que el consumo de CPU no será muy elevado.

La caché predictiva eleva este tiempo de actividad, puesto que no sólo se cargan los bloques que se necesitan inmediatamente, sino los colindantes, que potencialmente se solicitarán en un futuro cercano. Además, el *thread* de

cargas asíncronas tiene una sobrecarga adicional debida a la sincronización con los otros *threads* para solicitar las peticiones de cargas, sincronización entre *threads* de carga y entre múltiples clientes, protección de los *buffers* que pueden estar en uso por varios clientes, etc.

Estas diferencias en tiempo de actividad entre la caché predictiva y la reactiva se puede apreciar en las gráficas de la figura 5.15. En ellas se representa el tiempo de actividad normalizado entre 0 y 1 para cada máquina, por tamaño de *buffer* y por velocidad de vuelo.

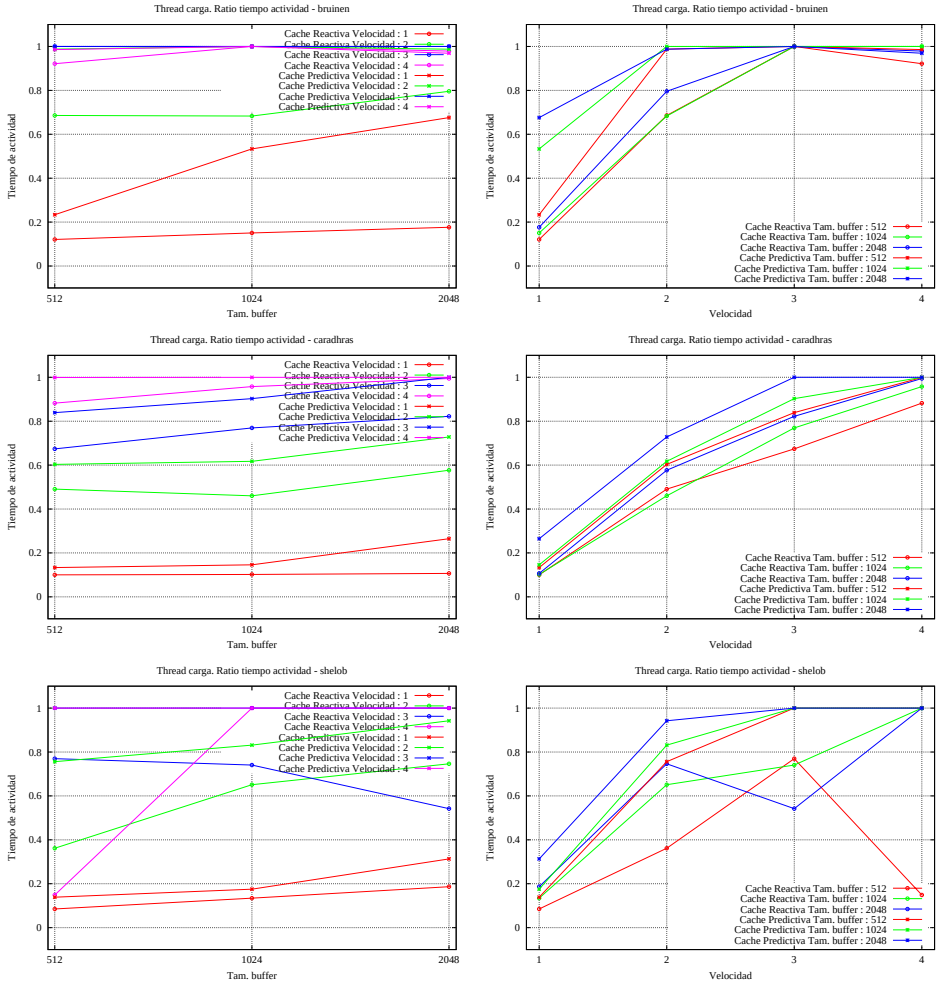


Figura 5.15: Ratio de actividad de los *threads* de cargas asíncronas, por tamaños de *buffer* y por velocidades de vuelo. Bruinen, Caradhras y Shelob.

Conclusiones de la prueba 1

Tras esta primera prueba comparativa entre las dos cachés implementadas en este trabajo, podemos resumir las siguientes conclusiones:

	Bruinen	Caradhras	Shelob
Carga	29,38 ms	5,64 ms	11,42 ms
Descompresión	17,36 ms	16,75 ms	8,65 ms

Cuadro 5.9: Tiempos medios de carga y descompresión.

- En velocidades altas, ningún sistema es bueno, la caché reactiva se muestra mejor en ocasiones debido a su simplicidad de funcionamiento.
- En velocidades normales[1] o bajas, el rendimiento de la caché predictiva es claramente superior.
- Los recursos de *hardware* disponibles son un factor importante a la hora de decidir. Cuando son muy limitados, puede ser más provechoso utilizar la caché reactiva por ser más ligera. Cuando el *hardware* es potente, el uso de la caché predictiva permite aprovechar las capacidades de dicho *hardware* para obtener un rendimiento superior.
- El tamaño de bloque 2048 es excesivamente grande y en general ofrece un rendimiento inferior y poco predecible, además de imponer un consumo de memoria notablemente mayor.
- La diferencia en el orden de carga de los niveles de detalle de la textura supone una diferencia de calidad a favor de la caché predictiva, que sin embargo se ve reflejada negativamente en los resultados cuantitativos de porcentaje de compleción de CacheTRAM.

5.3.3. Prueba 2. Compresión JPEG

En esta nueva prueba del sistema de caché de segundo nivel se utilizará la misma base de datos que en la prueba anterior, pero con un tamaño de bloque de 512×512 *texels* y con los bloques comprimidos en formato JPEG. En este caso el funcionamiento es similar, con la salvedad de que se añade el paso intermedio de descompresión de los *buffers* recibidos en memoria. Esta descompresión se realiza utilizando la CPU sobre el *buffer* en memoria principal.

Carga y descompresión de los *buffers*. Paralelización de tareas

En la tabla 5.9 se indican los tiempos medios de carga y descompresión de *buffers* en cada una de las máquinas de prueba. Como se puede apreciar, ambas tareas suponen unos tiempos bastante significativos y de dimensiones próximas.

[1] Entendiendo por velocidad normal aquella que permite al usuario apreciar el detalle mientras se mueve.

Sin embargo, existen diferencias entre las tareas que las hacen complementarias. La carga de *buffers* es en su práctica totalidad una operación de entrada/salida, y como tal no supone un consumo importante de CPU. Esto contrasta con la tarea de descompresión que realiza un uso intensivo de la CPU.

Puesto que se trata de procesos costosos y realizados de forma asíncrona, se pueden lanzar varios *threads* o hilos de ejecución con el fin de paralelizar estas operaciones.

En muchos casos, resulta más beneficioso realizar los accesos a disco en serie en lugar de simultáneamente. El acceso concurrente al disco puede llegar a degradar considerablemente la tasa de transferencia.

Dada la naturaleza de las dos tareas sí parece tener sentido paralelizar ambas, es decir, simultanear la carga de un *buffer* con la descompresión de otro. De esta forma se maximiza el uso tanto de los canales de E/S como de la CPU. Esto se ha implementado en la caché predictiva (CacheRAMPrecarga) mediante semáforos contadores. El usuario define tres parámetros de configuración para establecer el comportamiento de estos *threads*: número de *threads* de carga asíncrona, número de *threads* permitidos simultáneamente en operaciones de E/S y número de *threads* permitidos simultáneamente en operaciones de CPU. El primer parámetro indica a la caché predictiva cuántos *threads* debe lanzar. Los otros dos parámetros están en el cargador (CargadorRAM) utilizado en esta colección de datos, y sirven para inicializar dos semáforos contadores que restringen el acceso concurrente a las secciones de código encargadas de las operaciones de E/S (cargas) y CPU (descompresión).

En la implementación utilizada para estas pruebas, el cargador no permite operaciones de descompresión simultáneas debido a que la librería utilizada para el manejo de imágenes utiliza un estado global y no admite ejecuciones concurrentes (i.e., no es *thread-safe*), aunque esta limitación está previsto que se resuelva en la próxima versión de dicha librería.

Sin embargo, analizando los tiempos de las tareas, se puede concluir que no ofrecería demasiada ventaja tener más de un *thread* en descompresión (si se serializan las cargas), puesto que el tiempo de descompresión es similar o inferior al de carga.

Si se descarta la paralelización de cargas porque degradaría su eficiencia, en caso de tener varios *threads* descomprimiendo simultáneamente, éstos estarían esperando a que finalicen las cargas que les proveen de los datos a descomprimir. Esto cambiaría en caso de realizar otras operaciones o postprocesos adicionales en CPU a los *buffers* cargados, tras la descompresión. En este caso, si el tiempo de proceso en CPU se incrementa significativamente respecto al tiempo de carga, sí se obtendrían beneficios de la paralelización de estas operaciones de CPU.

	Threads	Velocidad 1	Velocidad 2	Velocidad 3
Caché Reactiva	1	62,71 %	44,24 %	4,93 %
Caché Predictiva	1	100,00 %	9,32 %	7,32 %
Caché Predictiva	2B	100,00 %	87,10 %	16,26 %
Caché Predictiva	2LIO	100,00 %	76,84 %	19,02 %
Caché Predictiva	3LIO	100,00 %	88,74 %	22,21 %
Caché Predictiva	4LIO	100,00 %	99,66 %	31,12 %

Cuadro 5.10: Porcentaje de aciertos de caché (Bruinen).

	Threads	Velocidad 1	Velocidad 2	Velocidad 3
Caché Reactiva	1	34,64 %	31,92 %	1,87 %
Caché Predictiva	1	100,00 %	99,50 %	2,00 %
Caché Predictiva	2B	100,00 %	100,00 %	3,53 %
Caché Predictiva	2LIO	100,00 %	100,00 %	3,86 %
Caché Predictiva	3LIO	100,00 %	100,00 %	4,10 %
Caché Predictiva	4LIO	100,00 %	100,00 %	5,56 %

Cuadro 5.11: Porcentaje de aciertos de caché (Caradhras).

Rendimiento de la caché

En las tablas 5.10 a 5.12 se muestran los porcentajes de aciertos de la caché en las diferentes configuraciones probadas, y para las velocidades 1 a 3. Se ha probado la caché reactiva con un *thread* (puesto que su implementación actual no admite más), y la caché predictiva con un *thread*, con dos *threads* limitando a una sóla carga simultánea, y dos, tres y cuatro *threads* sin límites en la concurrencia de cargas. En todos los casos, la operación de descompresión no admite acceso concurrente, debido a la limitación anteriormente mencionada.

Las gráficas de la figura 5.16 ilustran estos porcentajes en función de la velocidad para cada una de las máquinas de prueba.

Como resultado interesante, en la configuración de Bruinen se puede

	Threads	Velocidad 1	Velocidad 2	Velocidad 3
Caché Reactiva	1	39,89 %	39,49 %	15,77 %
Caché Predictiva	1	100,00 %	100,00 %	4,17 %
Caché Predictiva	2B	100,00 %	100,00 %	14,22 %
Caché Predictiva	2LIO	100,00 %	100,00 %	9,48 %
Caché Predictiva	3LIO	100,00 %	99,99 %	15,42 %
Caché Predictiva	4LIO	100,00 %	100,00 %	16,47 %

Cuadro 5.12: Porcentaje de aciertos de caché (Shelob).

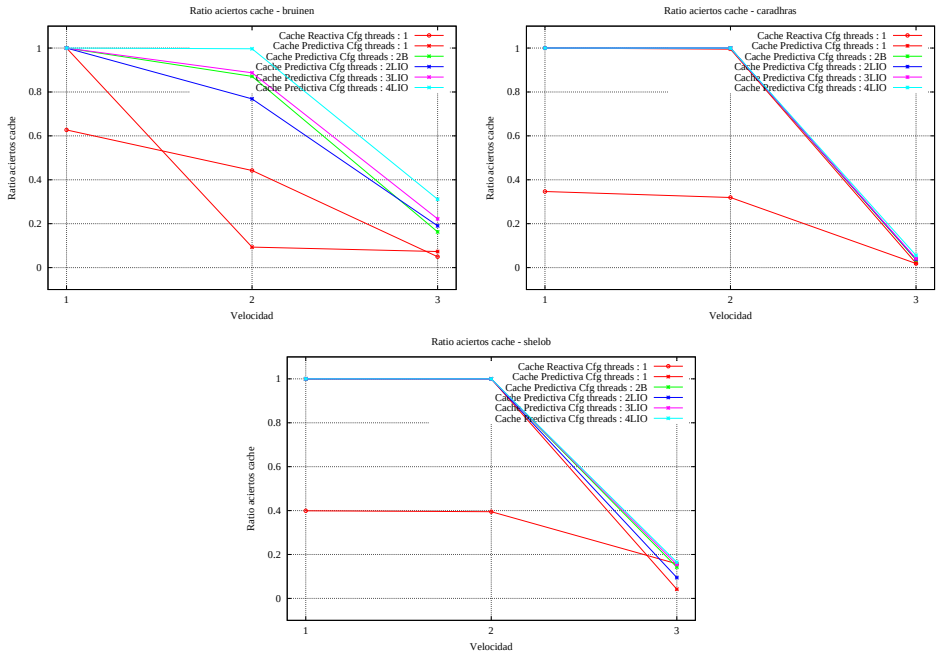


Figura 5.16: Ratio de aciertos según velocidad. Bruinen, Caradhras y Shelob.

apreciar que la desventaja de la caché predictiva frente a la reactiva en la velocidad 2 se ve compensada con el uso de múltiples *threads* de carga para simultanear la carga y la descompresión. Se observa también la ventaja del control de concurrencia en las cargas (configuración 2B en la gráfica), frente a la configuración libre (indicada como 2LIO en la gráfica).

En el caso de tres y cuatro *threads* de carga, se aprecia una cierta mejora sobre el uso de dos *threads*, pero esto puede resultar engañoso, puesto que por otra parte, este exceso de *threads* resta tiempo disponible al *thread* principal de *render* y actualización síncrona. Este problema se ve reflejado en los desbordamientos del tiempo de cuadro, que se incrementan considerablemente con el número de *threads*, especialmente en las configuraciones más pobres (ver gráfica 5.18). Las cargas simultáneas, sin embargo, no tienen un impacto tan grande en la tasa de transferencia como podría pensarse en un primer momento.

En las otras configuraciones, el comportamiento es similar, aunque el rendimiento es más próximo entre ellas. Se aprecia en cualquier caso una mejora ya en la velocidad 2 al utilizar varios *threads* de carga.

Nivel de actualización de la caché en TRAM y desbordamientos del tiempo de cuadro

El nivel de actualización de la caché en TRAM muestra unos resultados coherentes con el análisis hecho de las ratios de aciertos de caché (figura

5.17).

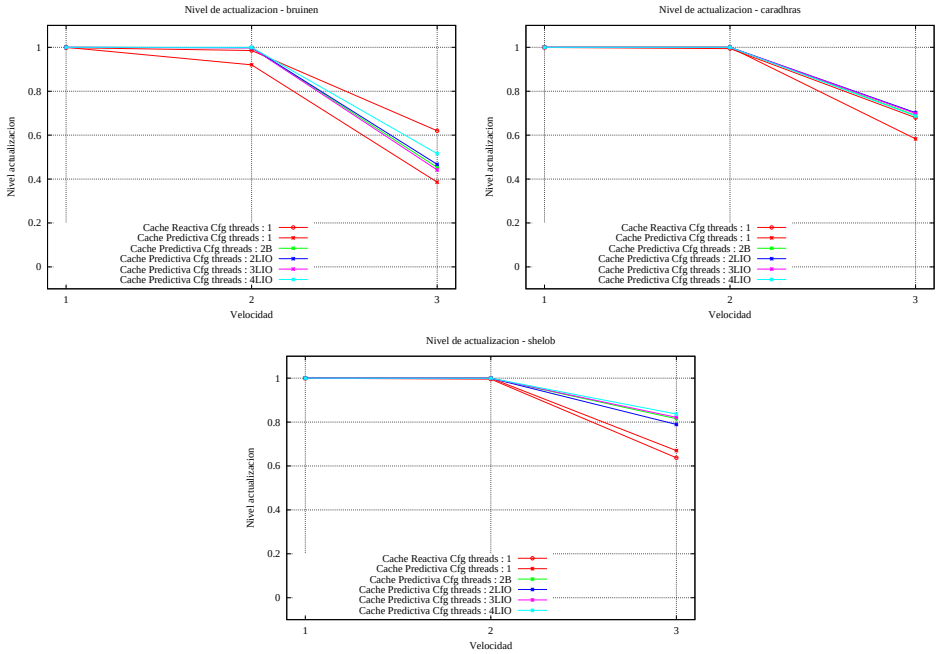


Figura 5.17: Nivel de actualización de la caché en TRAM según velocidad. Bruinen, Caradhras y Shelob.

Las gráficas de la figura 5.18 ilustran el impacto del uso de varios *threads* de carga sobre los desbordamientos del tiempo de cuadro.

Comparando estos resultados, podemos concluir que la configuración más eficiente es el la que utiliza dos *threads* sincronizados de forma que se alternen en las operaciones de E/S y CPU.

Gestión de las prioridades de los *threads*

Se ha considerado la opción de utilizar las características de tiempo real en la planificación de la ejecución de los *threads* (*real-time scheduling*) para asignar prioridades a los mismos. Por motivos de portabilidad, durante el desarrollo de este trabajo se ha utilizado el estándar IEEE POSIX 1003.1c (1995) para la creación y sincronización de los diversos hilos de ejecución. Este estándar define una serie de llamadas para que el sistema operativo realice una planificación priorizada de la ejecución de los *threads*. Sin embargo, durante las pruebas realizadas, se ha comprobado que las complicaciones del uso de la planificación en tiempo real superan a las ventajas.

En algunos campos de aplicación, denominados tiempo real duro (*hard real-time* [47]), las operaciones deben realizarse forzosamente dentro de un tiempo limitado (aunque esto no quiere decir que ese tiempo sea reducido), puesto que un retardo en el sistema puede tener consecuencias fatales.

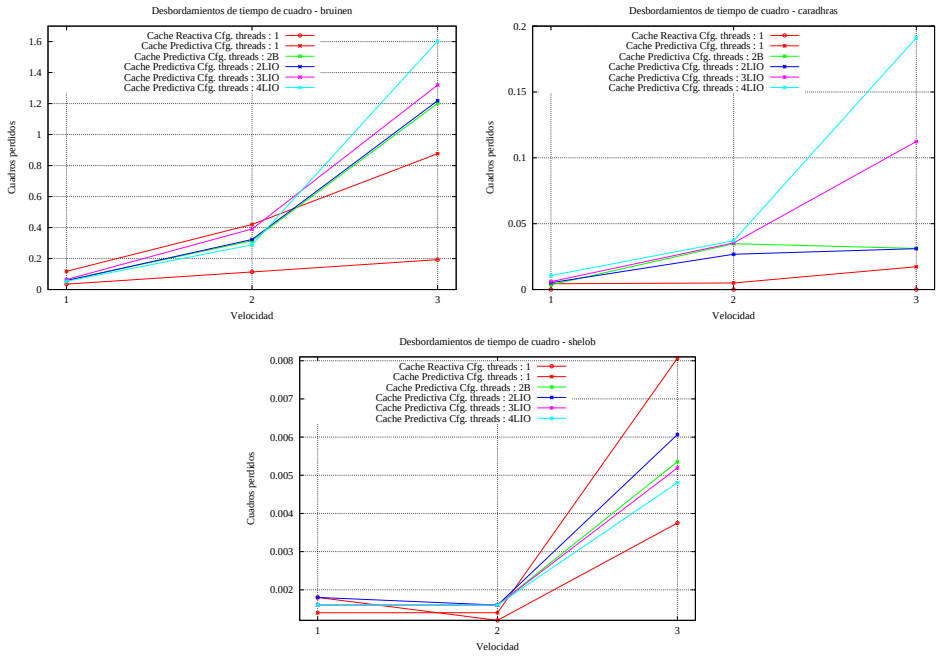


Figura 5.18: Desbordamientos del tiempo de cuadro según velocidad. Bruinen, Caradhras y Shelob.

Algunos ejemplos podrían ser el sistema de control de un reactor nuclear, de un sistema aeronáutico, o determinados sistemas médicos de asistencia a pacientes en estado crítico. En estos casos, no importa el esfuerzo que se requiera, es imprescindible garantizar que el tiempo destinado para una tarea no se desborde jamás.

Sin embargo, nuestro campo de aplicación cae dentro de los sistemas de tiempo real suaves (*soft real-time*), donde mantener la duración de las tareas críticas dentro de los límites establecidos es deseable e importante, pero las consecuencias de incumplirlo no son fatales.

El uso de los mecanismos para garantizar absolutamente el no desbordar el tiempo de cuadro resulta excesivamente costoso. El simple uso de las características ofrecidas por el estándar POSIX supone varios problemas:

- El uso de la planificación de tiempo real supone un coste importante para el núcleo del sistema operativo, de forma que el funcionamiento global será más lento que si no se usa. Se intercambia velocidad por ciertas garantías de finalización en el tiempo previsto, o más bien de que algunos hilos de ejecución no restan tiempo a otros con mayor prioridad.

- La sincronización de los *threads* también se complica, puesto que en los bloqueos de los datos accedidos desde varios *threads*, se producen condi-

ciones como la inversión de prioridad[47], que deben ser considerados y prevenidos en el código. Este aumento en la complejidad del código y en concreto de la sincronización, repercute a su vez negativamente en el rendimiento del sistema.

- El uso de prioridades elevadas puede interferir con tareas importantes del sistema operativo, como la captura de eventos de entrada del usuario, operaciones de E/S, etc. Por lo que se pueden producir eventualmente bloqueos del sistema.

- Problemas de portabilidad. Aunque el estandar POSIX define la API de las llamadas para la planificación de prioridades, los detalles concretos son dependientes de la plataforma y en todas la probadas se comporta de forma diferente. El uso de estas características afecta gravemente a la portabilidad de las aplicaciones, y especialmente las hace propensas a problemas como el mencionado en el punto anterior, de conflictos con otras tareas críticas del sistema operativo.

Una vez dicho esto, el uso de estas características tampoco garantiza al 100 % que las tareas será completadas en tiempo. Para esto sería necesario el uso de un sistema operativo de tiempo real, lo que quedaría fuera del entorno al que se orientan las aplicaciones del motor desarrollado, cuyo objetivo es integrarse dentro del entorno de trabajo normal de los usuarios de SIG.

En conclusión, es preferible un funcionamiento más fluido del sistema con alguna pérdida ocasional a una garantía de que el sistema irá exactamente a un rendimiento notablemente inferior.

Conclusiones de la prueba 2

A continuación se enumeran las conclusiones extraídas de esta segunda prueba.

- Esta prueba ha permitido verificar el correcto funcionamiento del sistema con conjuntos de datos comprimidos, tanto en disco local como a través de la red mediante el protocolo HTTP.

- El uso de compresión en los datos de origen incrementa la latencia del acceso a los datos por parte de la caché de primer nivel, lo cual hace más crítico el cuello de botella que de por sí supone la caché de segundo nivel. Sin embargo, resulta imprescindible en el caso de acceso a los datos a través de la red o bien cuando se manejan conjuntos de datos muy grandes y existen restricciones importantes en el espacio de almacenamiento disponible en el origen de datos.

- Para aliviar este cuello de botella, se han paralelizado las tareas de carga y descompresión de los buffers, probando diferentes configura-

ciones, que permiten concluir que la situación óptima combina dos hilos de ejecución alternando las tareas de carga y descompresión.

- El uso de mayor número de threads puede resultar beneficioso, especialmente cuando los recursos de *hardware* son elevados, pero aumenta el peligro de desbordamiento del tiempo de cuadro.

- La planificación de prioridades de ejecución de los *threads* se ha descartado tras observar que resulta completamente contraproducente, por los motivos ya expuestos.

5.4. Tratamiento de datos vectoriales

Los procesos de actualización estudiados hasta el momento en este capítulo corresponden a información de tipo *raster* en origen[2]. El proceso de actualización, cuando los datos originales están en formato vectorial, se trata de forma completamente diferente, tal y como ha sido descrito en la sección 4.6.

En esta sección se analizan los resultados de las diferentes opciones de optimización desarrolladas para determinar su impacto en el rendimiento de la actualización. El trabajo analizado en esta sección se centra en la optimización de los datos vectoriales, el preproceso que se había propuesto para realizar tanto *off-line* como bajo demanda durante la ejecución de la aplicación interactiva.

Para estos análisis se ha utilizado la colección de datos vectoriales procedente de información catastral (figura 5.19) descrita en la introducción del capítulo. Como ya se ha indicado, el módulo CacheRAM encargado de gestionar los datos vectoriales se basa en la librería OSG. Los datos originales, procedentes de un sistema CAD, han sido exportados a DXF y posteriormente se han convertido al formato binario nativo de OSG (.ive) mediante el cargador DXF que incluye OSG. Se utiliza este formato para el tratamiento más ágil de la información, puesto que los tiempos de carga se reducen notablemente. Los tamaños de fichero indicados más adelante se refieren a este formato (.ive).

El conjunto de datos original está compuesto de 1 141 224 vértices y 930 225 primitivas (líneas, polilíneas y cuadriláteros). Incluye 5 434 textos con nombres y números de calles.

[2]Entendiendo por "origen" los datos que entran al cargador de esta arquitectura (capa de acceso a datos en la figura 4.1). Esto no quiere decir que más allá de este punto los datos no puedan ser de origen vectorial. Un ejemplo sería un servidor WMS que genera las imágenes a partir de datos vectoriales. De cara a la GeoTextura, este servidor será una fuente de tipo *raster*, puesto que proporciona la información solicitada de esta manera.

Figura 5.19: Muestra del conjunto de datos vectoriales utilizados en las pruebas.

5.4.1. Descripción de la batería de pruebas

Las diferentes operaciones de optimización y combinaciones de ellas han dado lugar a diferentes versiones optimizadas del conjunto de datos original, cuyas características se describen a continuación.

Test 1 Conjunto de datos original, tal y como lo ha dejado el importador de OSG.

Test 2 Grafo de escena organizado en una estructura de árbol cuaternario adaptada a la configuración de teselas de la caché de GeoTextura. Se respetan los nodos del fichero original (no se subdividen primitivas), lo cual provoca la situación de que existen nodos con primitivas (líneas) dispersas por toda la escena que permanecen en un nodo cercano a la raíz, cuando esas primitivas se podrían distribuir en varios nodos, más próximos a las hojas del árbol cuaternario. El hecho de conservar los nodos del grafo de escena generados por el importador tiene la ventaja de que cualquier estado asociado a esos nodos se conserva (color, materiales, texturas, grosores de línea, etc). En la figura 5.20a se ilustra la estructura del grafo de escena en este test.

Test 3 Se realiza la misma estructuración jerárquica que en el test anterior, pero utilizando nodos LOD en lugar de grupos (figura 5.20b). Estos nodos LOD se configuran para determinar el nivel de detalle en función del tamaño en espacio pantalla (medido en pixels) y se establece un tamaño mínimo (10 pixels en esta prueba) por debajo del cual se

descartan. Este valor de umbral de visibilidad puede parecer elevado, pero no lo es tanto si se tiene en cuenta que lo que se mide es la proyección del volumen envolvente de todos los contenidos del nodo, que suele estar compuesto de varios elementos que individualmente tendrán un tamaño considerablemente inferior a esos 10 pixels.

Test 4 Se realiza el mismo proceso que en el test anterior pero todas las hojas colgadas de los grupos (nodos LOD) del árbol cuaternario se agrupan para que se realice una única comprobación en el LOD para todos ellos (figura 5.20c).

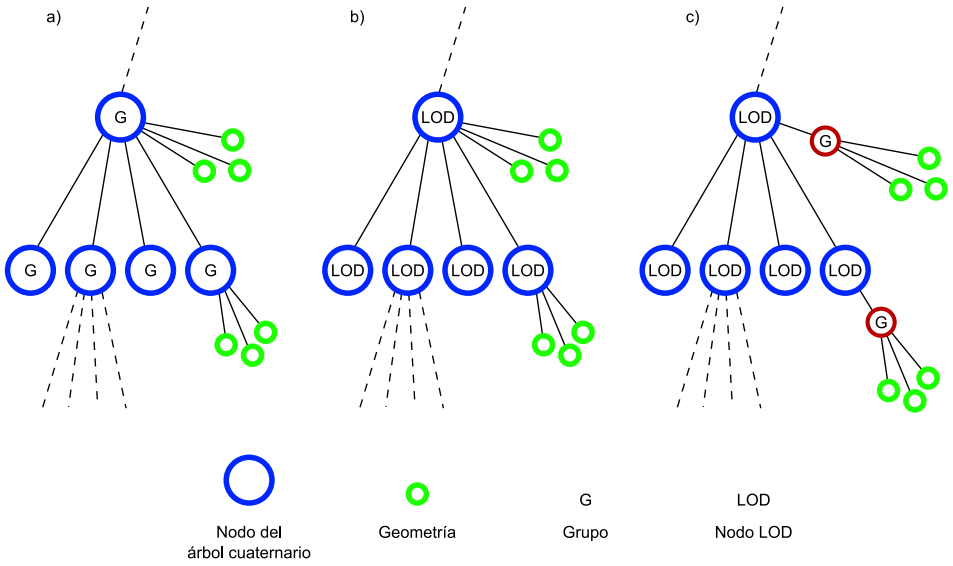


Figura 5.20: Estructura del scenegraph en las configuraciones de los tests 2, 3 y 4.

Test 5 Se realiza el mismo proceso que en el test anterior pero se desactiva el uso de *display lists* (opción por omisión) y VBOs (*vertex buffer objects*).

Test 6 Se realiza el mismo proceso que en el test anterior pero activando el uso de VBOs en lugar de *display lists*.

Test 7 Se descompone el grafo de escena a nivel de primitivas dibujables y se recompone utilizando el mínimo número de nodos organizados en una estructura óptima de árbol cuaternario adaptado a la configuración de teselas de la caché de GeoTextura. Se utilizan nodos LOD para los grupos del árbol cuaternario al igual que en los tests anteriores a partir del 3. Puesto que en este caso los elementos estarán más concentrados, se reduce el umbral que utilizan los nodos LOD para descartar elementos a 3 pixels. De esta forma se evita la desaparición de elementos visibles.

En este caso es importante tener en cuenta que es necesario agrupar las primitivas según estados (color, material, textura, etc.) y/o incluir

la información de color por vértice. El conjunto de datos utilizado en las pruebas no contiene información de color ni grosor ni ningún otro estado en las líneas, por lo que no hubo limitaciones a ese respecto. Sí se tuvo en cuenta, sin embargo, que los textos utilizan una textura con la tipografía utilizada, por lo que se respeta el estado (objeto *StateSet*) de los nodos que contienen texto[3].

En cada grupo del árbol cuaternario (nodo LOD), además de los cuatro LODs hijos correspondientes a los cuatro cuadrantes del siguiente nivel de árbol, se añade el mínimo número de geodas que sean necesarias: típicamente estarán separados los textos, con su estado correspondiente, y la geometría sin estados asociados. Se agrupan por tanto todas las primitivas, ya sean puntos, líneas o polígonos, en un único nodo para conseguir una escena óptima, siempre que no haya estados que obliguen a mantenerlas separadas.

Test 8 La escena optimizada es exactamente la misma que en el test anterior, pero en las pruebas se ha activado el suavizado de líneas.

Test 9 Se descompone el grafo de escena original en sus primitivas y se recompone igual que en el test 7, pero en este caso se convierten las primitivas de tipo línea y polilínea a polígonos para resolver los límites en anchura de línea y las incoherencias entre diferentes niveles de la textura. Estos problemas ya han sido descritos en la sección 4.6 del capítulo anterior. En esta prueba las primitivas poligonales utilizadas para la representación de las líneas han sido tiras de triángulos (`GL_TRIANGLE_STRIP`).

Test 10 La escena optimizada es exactamente la misma que en el test anterior, pero en las pruebas se ha activado el suavizado de polígonos.

Test 11 Se descompone el grafo de escena original en sus primitivas y se recompone en geometría poligonal, de la misma forma que en el test 9, pero utilizando cuadriláteros (`GL_QUADS`) para las líneas y tiras de cuadriláteros (`GL_QUAD_STRIP`) para las polilíneas. Este test se ha creado únicamente para comprobar si existen diferencias apreciables de rendimiento entre el uso de los dos tipos de primitiva y en caso de haberla cuál es mejor.

Test 12 La escena optimizada es exactamente la misma que en el test anterior, pero en las pruebas se ha activado el suavizado de polígonos.

En la tabla 5.13 se resumen las características de cada prueba y se muestra el tiempo de preproceso de datos vectoriales así como el tamaño de fichero

[3]Ese estado es único en toda la escena y está compartido por todos los nodos de texto.

Test	D	QT	LOD	DL	VBO	Prims	AA	T	Tamaño
1				x				-	24 057 831
2		x		x				13,01 s	24 127 395
3		x	10	x				13,17 s	24 711 611
4		x	10	x				13,19 s	24 638 825
5		x	10					13,37 s	24 638 825
6		x	10		x			14,15 s	24 638 825
7	x	x	3	x				13,25 s	28 340 474
8	x	x	3	x			x	13,25 s	28 340 474
9	x	x	3	x		TS		18,78 s	42 187 424
10	x	x	3	x		TS	x	18,78 s	42 187 424
11	x	x	3	x		Q/QS		18,78 s	40 834 394
12	x	x	3	x		Q/QS	x	18,78 s	40 834 394

Cuadro 5.13: Configuraciones de test de datos vectoriales. D: descomposición de primitivas, QT: árbol cuaternario (*quadtree*), LODs: umbral de visibilidad en pixels/*texels*, DL: uso de *display lists*, VBO: uso de *vertex buffer objects*: Prims=tipo de primitiva para líneas (TS=tristrips, Q=quads, QS=quadstrips), AA: suavizado de puntos/líneas/polígonos, T: tiempo de preproceso en segundos, Tamaño: tamaño del fichero resultante en bytes.

resultante. El tiempo de generación del grafo de escena optimizado se ha medido en la máquina más modesta (Bruinen) y se considera sin tener en cuenta el tiempo de carga de la escena original, que en esa máquina asciende a 13,68 segundos.

Los parámetros analizados en cada una de las pruebas han sido los siguientes:

- Tiempo de regeneración completa de la caché.
- Tiempo de regeneración por tesela.
- Tiempos de las diferentes fases del render del grafo de escena de OSG correspondiente a los datos vectoriales de CargadorRAMSceneGraphOSG. En concreto se han medido los tiempos de las fases *event*, *update*, *cull* y *draw*, así como el tiempo consumido por la GPU.

5.4.2. Resultados de las pruebas

Los resultados del **test 1**, que corresponde al conjunto de datos tal cual se obtiene del cargador de DXF, se toman como referencia para evaluar las otras técnicas. Como se puede observar en las gráficas 5.21 a 5.23, todas ellas mejoran su rendimiento en todos los casos salvo contadas excepciones cuando el tamaño de tesela corresponde con el tamaño de ventana.

En las tablas 5.15 a 5.17 se ha utilizado un tamaño de tesela intermedio de 256×256 *texels* para analizar el efecto de las diferentes pruebas en los tiempos de generación de tesela. Es importante no sólo observar los promedios de tiempos de tesela sino muy especialmente la variación de tiempos y los mínimos y máximos alcanzados en cada caso. Como ya se indicó en el capítulo anterior, la predictibilidad del tiempo de generación de las teselas es tan importante como el propio tiempo.

Como se aprecia claramente en las gráficas y las tablas, la organización espacial jerárquica de los contenidos de la escena en el árbol cuaternario (**test 2**) es el factor que mayor impacto tiene en el rendimiento de la generación de las teselas a partir de los datos vectoriales. En concreto se reduce notablemente el tiempo de la fase *cull* del render de la escena OSG, aproximándolo al tiempo de la fase *draw*. En la figura 5.24 se muestran los tiempos de las diferentes fases para los dos primeros tests.

El efecto del uso de nodos LOD en el **test 3** produce efectos diferentes según la configuración utilizada.

En las dos máquinas con Windows (Bruinen y Shelob) se aprecia una reducción del tiempo medio y de la desviación típica, con lo que se mejora tanto en rendimiento global como en la predictibilidad tan importante para minimizar los desbordamientos del tiempo de cuadro. Se aprecia también un muy leve aumento en los máximos del tiempo de generación de tesela, fruto de la sobrecarga impuesta por el funcionamiento de los nodos LOD.

En la configuración con Linux, en cambio, se produce un leve empeoramiento respecto al test anterior en cuanto a media y desviación típica, y considerablemente más grave en cuanto al tiempo máximo de generación de tesela.

El efecto positivo de los nodos LOD se aprecia especialmente cuando el tiempo de la fase *draw* excede al tiempo de la fase *cull*, lo cual en el ejemplo estudiado no se produce, como se puede ver en la figura 5.24. Esta condición es absolutamente dependiente de los datos visualizados y su distribución espacial. Se puede considerar que este ejemplo es una de las peores situaciones posibles, por lo que es esperable que en otros conjuntos de datos el comportamiento del uso de nodos LOD sea igual o superior.

La agrupación de los nodos hoja del árbol de LODs (**test 4**) produce una mejora notable en todos los casos con una excepción (Shelob con tamaño de ventana 1024). En este caso se ve incrementado el tiempo máximo y con él la desviación típica, sin embargo el valor de tiempo medio de generación de tesela sigue mostrándose mejor.

Los **tests 5 y 6** desactivan el uso de *display lists* (comportamiento por omisión de OSG, utilizado en los demás tests) y activan el uso de VBOs respectivamente. En ambos tests y en todas las configuraciones y tamaños de ventana analizados se produce una leve caída del rendimiento respecto al test anterior, por lo que se descartan estos modos de funcionamiento en favor del modo por omisión de OSG, el uso de *display lists*.

Test	Vértices	Primitivas	Geodas	Grupos	LODs
4	1 141 224	930 225	64 346	475	1 827
7	1 141 224	930 225	53 669	0	76 719
9	2 182 384	1 835 434	53 669	0	76 719
11	2 182 384	930 225	53 669	0	76 719

Cuadro 5.14: Número de nodos, vértices y primitivas de los tests 7, 9 y 11.

El **test 7** supone un cambio en la filosofía de trabajo. En lugar de respetar los nodos del grafo de escena se descomponen en sus primitivas dibujables (en este caso líneas, polilíneas y textos) y se recompone un nuevo grafo de escena agrupando estos elementos en los nodos del árbol cuaternario.

La escena optimizada de esta forma se compara con el test 4, el que muestra el mejor comportamiento de los analizados hasta el momento. En esta comparación se aprecia que el rendimiento es claramente superior en todos los aspectos (media, desviación típica, mínimo y máximo) llegando en algunos casos a la mitad de tiempo.

Esta mejora tan notable ha de considerarse sin embargo con cautela, puesto que como ya se ha mencionado, en caso de que los nodos del grafo de escena original tengan estados diferentes (material, textura, etc), existirían limitaciones a la hora de estructurar la escena como se ha hecho en este caso.

Los tests analizados hasta ahora utilizaban primitivas de tipo línea, las cuales tienen una limitación de tamaño que ya ha sido descrita en la sección 4.6 donde se planteaba la solución de convertirlas a polígonos dándoles un grosor fijo en espacio mundo. Esta conversión se ha realizado en los **tests 9 y 11**. Se han utilizado diferentes primitivas de OpenGL en ambos casos (`GL_TRIANGLE_STRIP` en el test 9 y `GL_QUADS` y `GL_QUAD_STRIP` en el test 11) para observar posibles diferencias en el rendimiento debidas a este factor.

Las diferencias observadas entre los tests 7, 9 y 11 son mínimas, con una leve ventaja del test 7, lo cual tiene sentido teniendo en cuenta que los tests 9 y 11 doblan el número de vértices de la geometría dibujada (ver tabla 5.14). La diferencia de rendimiento entre las tiras de triángulos y de cuadriláteros muestran una cierta tendencia en favor de las primeras, si bien esta diferencia es muy reducida (del orden de decenas de μs).

Los tests *8, 10 y 12* utilizan las mismas escenas que los 7, 9 y 11 respectivamente, con el suavizado de líneas (en el caso del test 8) o de polígonos (tests 10 y 12) para reducir el efecto de *aliasing*. Como se puede observar en las tablas de tiempos, las diferencias entre la activación o no activación del suavizado de líneas o polígonos no sólo no empeora significativamente al rendimiento sino que en ocasiones ofrece un rendimiento mejor. Se puede observar sin embargo en las figuras 4.27 y 4.28 que visualmente sí se aprecian diferencias notables en la calidad.

Test	T. ventana	Media	D. típica	Mínimo	Máximo
1	512	23,09	12,48	15,38	90,21
2	512	8,22	8,08	1,14	30,56
3	512	7,08	6,71	1,25	33,67
4	512	6,31	6,52	0,57	30,42
5	512	10,20	8,53	2,75	42,84
6	512	6,88	7,30	0,61	42,77
7	512	4,42	4,58	0,38	14,30
8	512	4,54	5,16	0,35	54,03
9	512	4,38	4,54	0,36	14,63
10	512	4,40	4,55	0,35	14,63
11	512	4,41	4,56	0,34	14,83
12	512	4,40	4,51	0,38	14,51
1	1024	20,39	12,18	15,08	106,44
2	1024	4,64	5,93	0,60	30,49
3	1024	4,45	5,24	0,58	36,73
4	1024	3,81	5,13	0,52	30,82
5	1024	7,36	7,43	1,27	58,95
6	1024	4,48	6,31	0,57	53,81
7	1024	2,33	3,24	0,34	15,31
8	1024	2,38	3,27	0,33	14,55
9	1024	2,45	3,31	0,35	14,61
10	1024	2,47	3,33	0,36	14,57
11	1024	2,46	3,36	0,34	17,26
12	1024	2,47	3,32	0,34	14,65

Cuadro 5.15: Tiempos (en ms) de generación de tesela en las diferentes pruebas (Bruinen).

Test	T. ventana	Media	D. típica	Mínimo	Máximo
1	512	56,83	30,54	38,89	211,68
2	512	19,37	18,89	2,64	76,38
3	512	19,78	20,81	2,68	101,45
4	512	17,14	18,85	1,53	91,37
5	512	25,74	28,16	3,17	140,21
6	512	26,81	31,36	1,65	154,56
7	512	12,18	13,14	0,89	42,28
8	512	12,13	13,07	0,89	41,95
9	512	12,21	13,21	0,89	45,83
10	512	12,40	13,40	0,90	43,84
11	512	12,24	13,23	0,90	43,21
12	512	12,27	13,25	0,90	43,42
1	1024	50,77	29,49	34,54	258,09
2	1024	11,22	14,21	1,40	76,33
3	1024	12,38	16,38	1,40	101,33
4	1024	10,68	14,99	1,21	91,39
5	1024	16,20	22,27	1,66	140,21
6	1024	16,19	24,82	1,32	154,58
7	1024	6,56	9,44	0,84	41,81
8	1024	6,55	9,44	0,84	41,70
9	1024	7,02	9,91	0,88	42,99
10	1024	7,06	9,95	0,89	43,23
11	1024	7,06	9,96	0,88	43,34
12	1024	7,05	9,92	0,88	43,14
1	2048	45,71	26,89	27,32	304,24
2	2048	6,50	9,62	0,82	76,51
3	2048	7,27	11,47	0,81	104,16
4	2048	6,10	10,58	0,81	91,41
5	2048	9,52	15,66	0,80	139,90
6	2048	8,83	17,41	0,81	154,66
7	2048	3,69	6,25	0,58	41,95
8	2048	3,71	6,27	0,57	42,06
9	2048	3,95	6,69	0,58	43,38
10	2048	3,95	6,67	0,58	43,14
11	2048	3,95	6,70	0,58	43,54
12	2048	3,94	6,66	0,58	43,22

Cuadro 5.16: Tiempos (en ms) de generación de tesela en las diferentes pruebas (Caradhras).

Test	T. ventana	Media	D. típica	Mínimo	Máximo
1	512	15,69	7,03	11,03	51,33
2	512	5,31	5,42	0,40	20,05
3	512	4,21	4,26	0,41	20,35
4	512	4,00	4,22	0,30	19,71
5	512	6,22	5,72	1,11	28,35
6	512	4,56	4,96	0,31	23,90
7	512	2,66	2,77	0,22	8,98
8	512	2,55	2,67	0,23	8,87
9	512	2,68	2,81	0,21	9,31
10	512	2,55	2,69	0,23	9,19
11	512	2,67	2,80	0,21	9,35
12	512	2,69	2,85	0,22	15,32
1	1024	13,96	6,75	10,74	59,38
2	1024	2,81	3,96	0,31	19,88
3	1024	2,48	3,29	0,31	19,99
4	1024	2,42	3,43	0,25	26,04
5	1024	3,73	4,57	0,47	29,46
6	1024	2,71	3,92	0,27	27,75
7	1024	1,39	1,98	0,21	9,06
8	1024	1,38	2,01	0,21	13,27
9	1024	1,48	2,07	0,21	9,30
10	1024	1,45	1,98	0,23	9,24
11	1024	1,48	2,07	0,21	9,30
12	1024	1,48	2,06	0,21	9,31
1	2048	12,67	6,20	7,86	71,21
2	2048	1,46	2,60	0,20	19,90
3	2048	1,37	2,24	0,20	20,26
4	2048	1,27	2,22	0,20	19,93
5	2048	2,49	3,27	0,18	36,93
6	2048	1,50	2,90	0,18	31,25
7	2048	0,76	1,26	0,17	10,10
8	2048	0,77	1,31	0,17	14,29
9	2048	0,80	1,30	0,17	9,20
10	2048	0,84	1,54	0,17	19,51
11	2048	0,82	1,41	0,16	12,13
12	2048	0,81	1,35	0,17	12,14

Cuadro 5.17: Tiempos (en ms) de generación de tesela en las diferentes pruebas (Shelob).

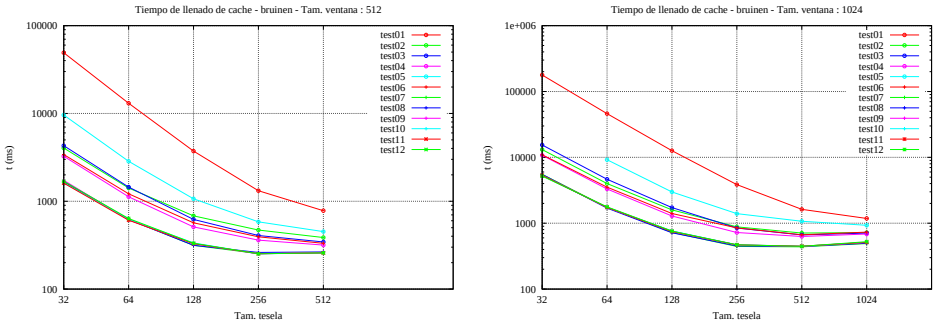


Figura 5.21: Tiempo de regeneración de la caché completa (Bruinen). Tamaños de ventana: 512 y 1024.

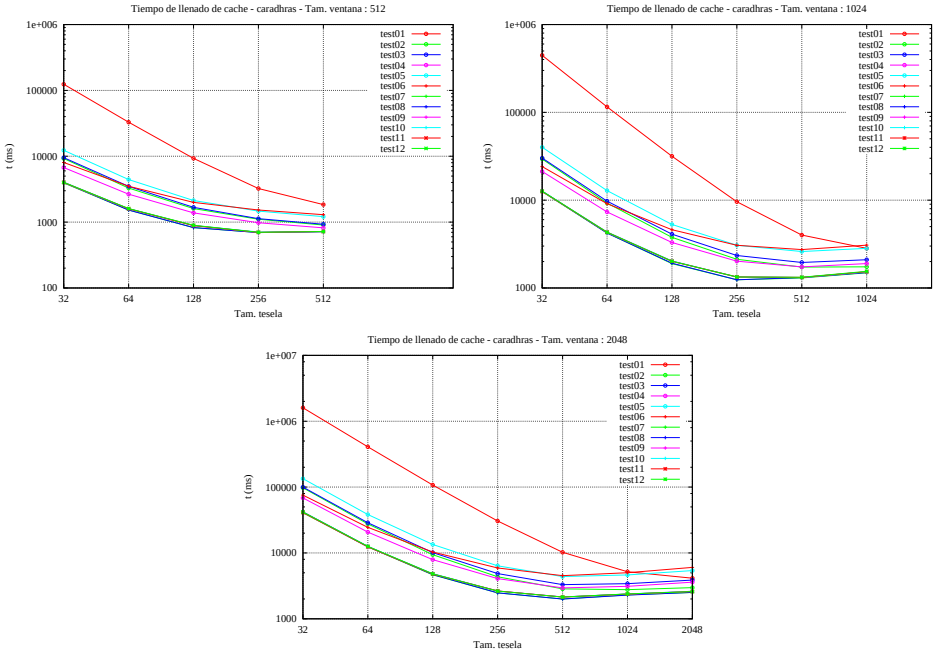


Figura 5.22: Tiempo de regeneración de la caché completa (Caradhras). Tamaños de ventana: 512, 1024 y 2048.

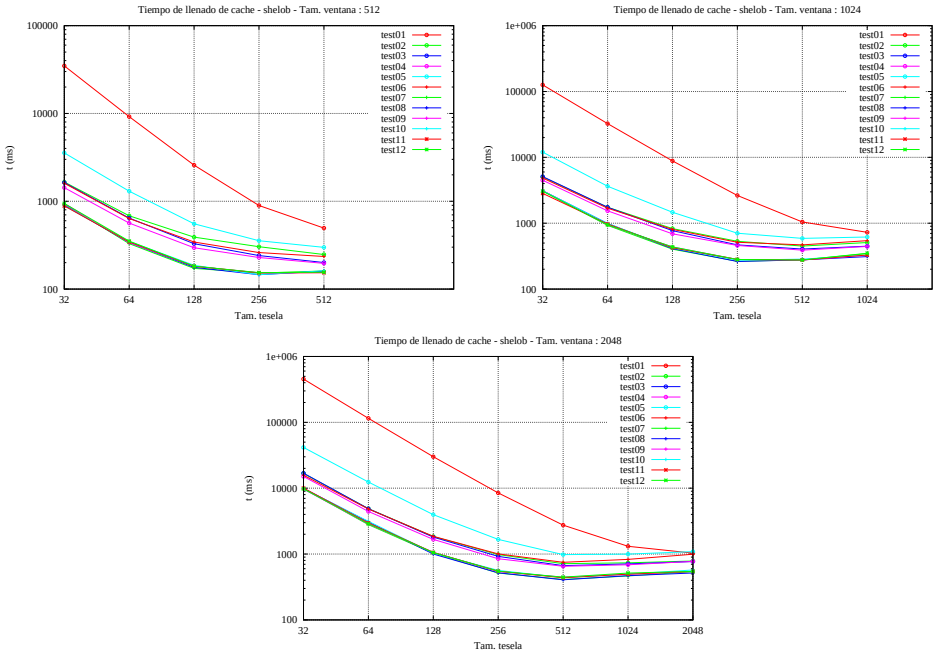


Figura 5.23: Tiempo de regeneración de la caché completa (Shelob). Tamaños de ventana: 512, 1024 y 2048.

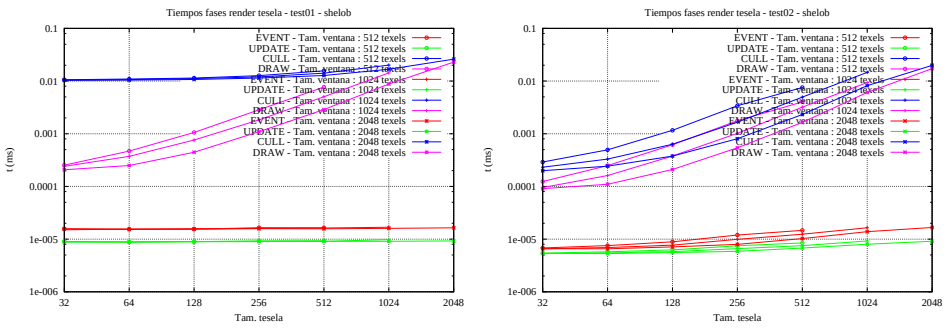


Figura 5.24: Tiempos medios de las diferentes fases de OSG en la generación de una tesela. Tests 1 y 2.

5.4.3. Conclusiones

Respecto al tratamiento de datos vectoriales, tras el análisis realizado sobre los resultados de las doce pruebas descritas, se pueden extraer algunas conclusiones claras.

- La técnica de optimización que produce una mejora más drástica en el rendimiento es sin duda el uso de la estructura jerárquica de árbol cuaternario adaptada a las teselas de la caché, acercándose en algún caso la diferencia de rendimiento a un orden de magnitud. Esto se aprecia especialmente en la mejora de los tests 7 al 12, en los que las primitivas se distribuyen de manera óptima en el árbol cuaternario.

- La opción que ofrece mejor resultado conservando los nodos del grafo de escena original es el test 4, que utiliza el árbol cuaternario de nodos LOD agrupando los elementos visibles de cada grupo del árbol y activa el uso de *display lists* para el *render* de las teselas.

- El mejor rendimiento se obtiene descomponiendo los nodos del grafo de escena y distribuyendo las primitivas en los nodos del árbol cuaternario más profundos que contengan completamente esas primitivas.

- La conversión de las líneas a polígonos para poder usar un grosor fijo en espacio mundo (espacio textura) no supone impacto notable sobre el rendimiento.

- El uso de suavizado de líneas y polígonos mediante los estados `LINE_SMOOTH` y `POLYGON_SMOOTH` de OpenGL mejoran notablemente la calidad de la visualización sin provocar un impacto apreciable en el rendimiento.

5.5. Ejemplos de aplicación

Tras el análisis, principalmente cuantitativo, de los diferentes aspectos clave del sistema desarrollado, para finalizar este capítulo de resultados se repasarán algunas de las aplicaciones donde está actualmente en uso el motor GeoTextura.

5.5.1. Demostrador SIG 3D genérico

Una de las primeras aplicaciones prácticas de los trabajos desarrollados en esta tesis doctoral consiste en un demostrador genérico del Sistema Avanzado de Navegación sobre Terreno Interactivo (SANTI), sobre el cual se muestra información de diversos tipos: tanto modelos 3D como información 2D de tipo *raster* y vectorial.

Configuración de las texturas virtuales

Esta aplicación combina tres texturas virtuales:

- **Ortofotografía** aérea de Galicia, procedente de las imágenes del SIGPAC. La resolución de la textura es de $1\,048\,576 \times 1\,048\,576$ *texels* para una zona de aproximadamente 213×225 kilómetros. El formato es RGB, originalmente de 24 bits por *texel*, pero para reducir el uso de memoria y disco y la transferencia por el bus a la sexta parte, se utiliza el formato comprimido S3TC DXT1.

- Información **vectorial estática**. En esta textura se combina una amplia colección de capas SIG vectoriales de diversos tipos (puntos, líneas, polígonos y textos), y relativos a diferentes temas (evolución histórica de las zonas pobladas, red de aguas, datos catastrales, límites administrativos, hidrografía, espacios protegidos, etc.), con fines meramente demostrativos. La resolución en *texels* y la extensión geográfica de esta textura virtual coinciden con las de la ortofotografía anterior, pero el formato de *texel* es de cuatro canales (RGBA8888), lo cual incrementa el uso de memoria en un tercio. Sin embargo, es necesario este canal alfa para poder combinar las texturas correctamente. Además, al tratarse de una textura vectorial (las teselas se generan mediante *render* a textura), no se puede utilizar un formato comprimido.

- Información **vectorial dinámica**. Esta textura virtual contiene información vectorial actualizada con una frecuencia de un segundo. En este caso, la información consiste en una simulación de la evolución en el tiempo de la densidad del tráfico en las carreteras de una zona reducida. Dado que la extensión geográfica de la textura es menor, su resolución se ha reducido en consecuencia a 65536×65536 *texels*. El formato de *texel* es el mismo que en la textura anterior.

El hecho de separar las dos texturas vectoriales (estática y dinámica), permite manejar un gran volumen de información estática, combinada con la información dinámica (cuyas teselas se regeneran con elevada frecuencia), sin afectar de manera notable al rendimiento ni a la calidad de la visualización. Además, la información dinámica está situada en un área más pequeña, por lo que las texturas tienen diferentes extensiones y resoluciones.

El tamaño de ventana utilizado para las texturas se sitúa entre 1024 y 2048 *texels* de lado, según la disponibilidad de memoria en la tarjeta gráfica. Con la ventana de 1024, las texturas virtuales consumen exactamente 100 MBytes (20+48+32).

El tamaño de bloque de la ortofotografía es de 512×512 *texels*, y los tamaños de tesela de las tres texturas son de 512, 256 y 256 respectivamente.

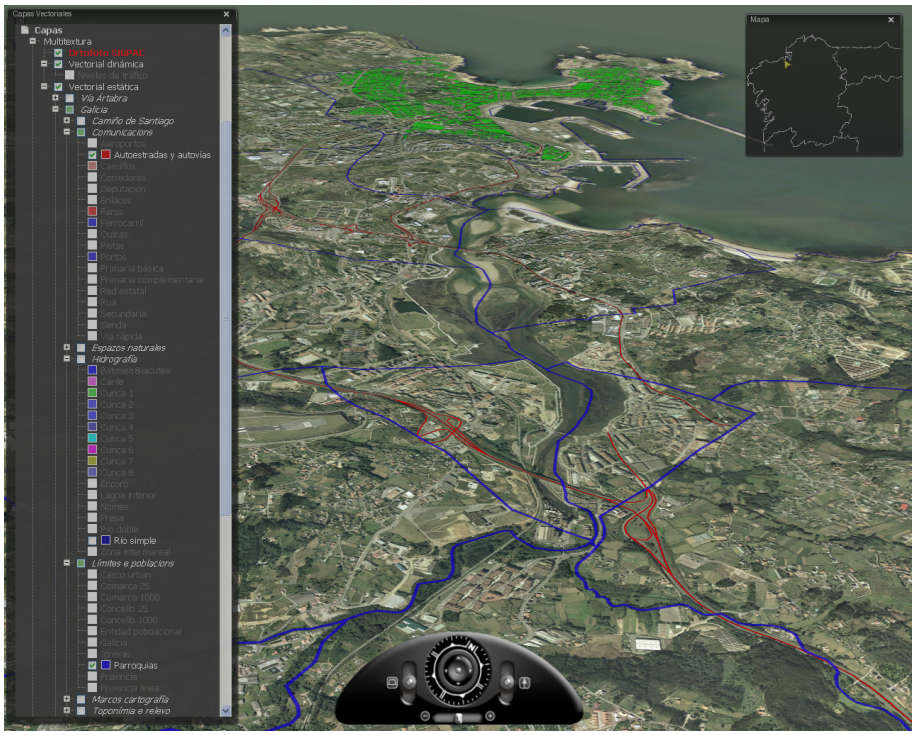


Figura 5.25: Demostrador genérico de SANTI y GeoTextura. Ortofotografía mezclada con la información vectorial.

Combinación de las texturas virtuales

Las tres texturas descritas se combinan en el *fragment shader* definido para la multitextura, que sigue el comportamiento que se describe a continuación.

Se toma como partida un color base para la geometría del terreno, que se sustituye por la primera textura (ortofotografía) en caso de estar activada.

Sobre este color sólido o la ortofotografía, se aplican la segunda y tercera textura en caso de estar activas, mediante la operación habitual de fundido por canal alfa (operador `mix()` de GLSL) (figura 5.26).

Las coordenadas de textura se calculan automáticamente a partir de la posición de los vértices de la geometría en el *vertex shader* asociado a la multitextura.

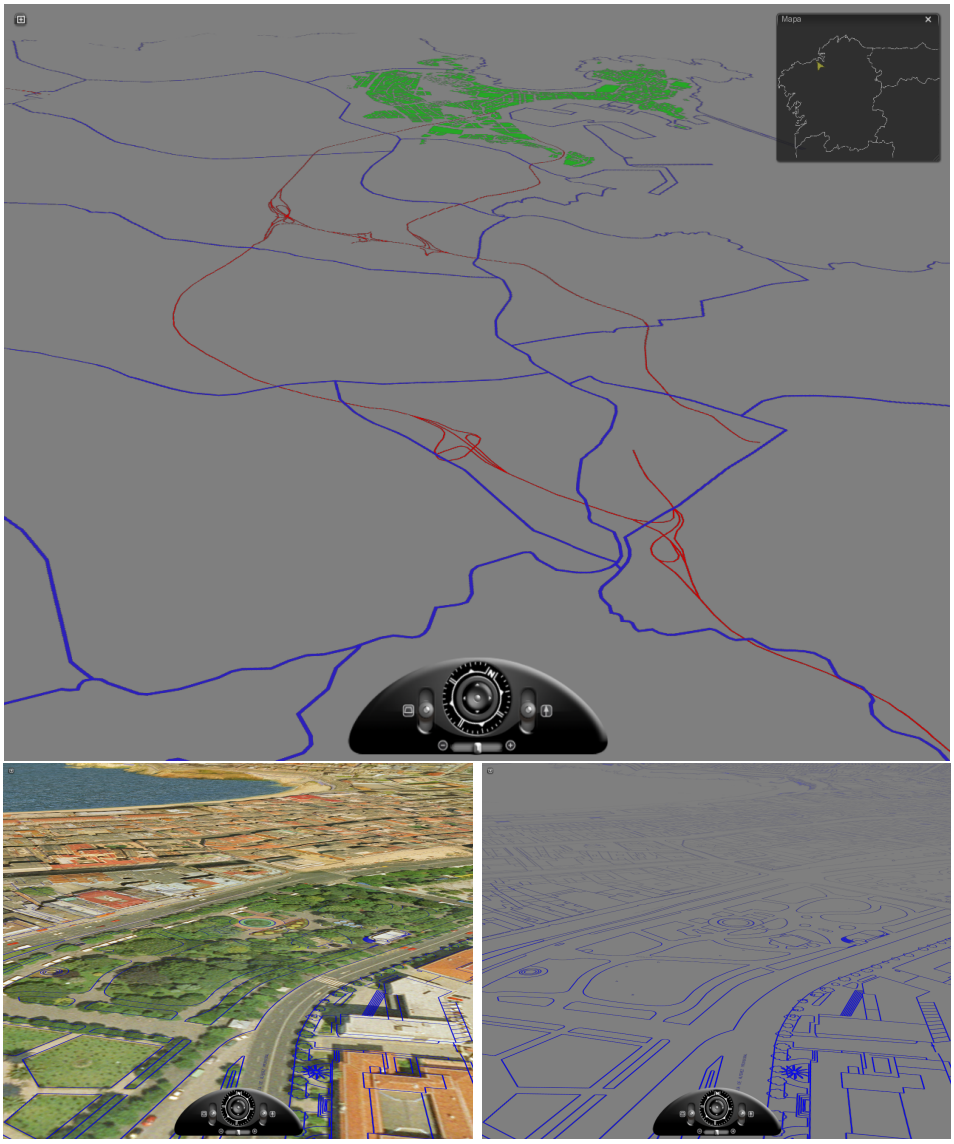


Figura 5.26: Demostrador genérico de SANTI y GeoTextura. Información vectorial sobre la ortofotografía y sobre un color sólido de fondo.

Interfaz de selección de capas

Para ofrecer una interfaz de usuario cómoda y potente, se ha desarrollado un sistema genérico de selección de capas, ilustrado en la figura 5.27, que permite al usuario activar o desactivar capas y cambiar atributos como color, grosor de líneas y puntos, etc.

Este sistema se basa en dos clases abstractas: `GestorCapasGIS` y `CapaGIS`. Estas clases abstractas se complementan con un gestor de interfaz gráfica de usuario que maneja de forma transparente cualquier módulo del sistema que implemente la interfaz definida en dichas clases.

La interfaz gráfica permite agrupar jerárquicamente toda la información SIG, de forma que se puede organizar visualmente de manera cómoda y activar o desactivar colecciones de datos con una sola orden.

También permite gestionar secuencias temporales de texturas, poniendo en marcha la reproducción automática, variando la velocidad, avanzando o retrocendiendo en el tiempo, o activando un instante concreto. Esto resulta de gran utilidad para las texturas dinámicas desarrolladas en este trabajo. Un par de ejemplos utilizados de este tipo de textura son la secuencia de los doce meses del año 2004 de la colección de imágenes de Blue Marble[116] y la secuencia de ortofotografías de la ciudad de A Coruña a lo largo de varios años.

Este esquema de gestión de capas SIG se ha utilizado en diferentes partes del sistema desarrollado:

- Multitextura. Para permitir mostrar u ocultar fácilmente las texturas virtuales disponibles en una multitextura.
- Cargador de capas SIG vectoriales. Los cargadores como `CargadorRAMScenegraphOSGGIS` o `CargadorRAMWMS`, para permitir la activación, desactivación y manipulación del aspecto de las capas vectoriales disponibles.
- Capas SIG. A su vez, las capas SIG de los cargadores mencionados en el punto anterior, pueden contener entidades SIG con identidad propia[4]. En estos casos, una capa SIG será a la vez una capa del cargador y un gestor cuyas capas son las entidades SIG a las que se da acceso de forma individual en la interfaz gráfica.
- Secuencias temporales. Para aquellas texturas virtuales dinámicas que se compongan de colecciones de datos temporales, se permite el acceso directo a cada uno de los estados o instantes en el tiempo de la textura.

[4] El ejemplo más habitual son las capas en formato *shape* de ESRI, que están compuestas por entidades que tienen información asociada en una tabla de una base de datos relacional. El hecho de conservar la independencia de estas entidades, además de ampliar las posibilidades de visualización discriminando exactamente qué elementos se desean mostrar y cuáles no, permite acceder a la información propia de cada entidad.

Figura 5.27: Interfaz de selección y configuración de capas SIG.

Conclusiones

La aplicación descrita, además de su fin práctico como demostrador para la captación de nuevos proyectos, ha supuesto un excelente banco de pruebas para validar el sistema desarrollado y su facilidad de integración dentro de un sistema de visualización de terreno ya existente, así como su interoperabilidad con otros componentes.

También ha servido para desarrollar y probar la interfaz gráfica de selección y manipulación de capas descrita.

En muchos casos, esta configuración genérica se utiliza como plantilla para la creación y configuración de nuevas aplicaciones basadas en SANTI y GeoTextura.

5.5.2. Sistema integral de gestión de tráfico

El sistema integral de gestión de tráfico es una aplicación para la monitorización y control de los dispositivos utilizados por el Centro de Gestión de Tráfico de la zona noroeste de España. Este centro, perteneciente a la Dirección General de Tráfico, dispone de una red de comunicaciones que conecta dicho centro con una serie de dispositivos entre los que se incluyen estaciones meteorológicas, paneles de mensaje variable, medidores de aforo de tráfico, cámaras de televisión y radares fijos.

El objetivo de la aplicación es proporcionar una interfaz rápida y cómoda donde ubicar geográficamente estos elementos sobre una vista 3D realista del terreno. Esto permite a los operadores del centro un análisis mucho más profundo, completo y ágil que con las interfaces que manejaban anteriormente.

En la versión actual se encuentra desarrollada la conexión con las estaciones meteorológicas y los paneles de mensaje variable.

En el primer caso se muestra la información de manera gráfica en 3D y a través de fichas 2D que ofrecen acceso a la totalidad de los datos obtenidos de las estaciones. Se controlan también determinadas condiciones de alarma y se permite de forma inmediata trasladar la cámara a la posición de cualquier equipo o del lugar donde se haya producido una alarma.

Los paneles de mensaje variable se visualizan de forma simbólica a gran escala y también ubicados exactamente en la posición del terreno y con su forma, escala y contenidos reales. El contenido de los paneles se actualiza en tiempo real y la aplicación permite variarlos a través de una completa interfaz de usuario.

Configuración de texturas virtuales

En este caso no se muestra información de tipo vectorial sobre el terreno, sólo se utilizan dos texturas virtuales de tipo *raster* que se pueden activar independientemente o combinadas.

- Ortofotografía aérea de la zona noroeste de España, Galicia y parte de Castilla y León (hasta Benavente), coincidiendo con la zona geográfica competencia del Centro de Gestión de Tráfico del Noroeste. Las imágenes han sido tomadas del SIGPAC y del PNOA, la resolución de la textura es de $524\,288 \times 524\,288$ *texels*. El formato utilizado es RGB888 (24 bits por *texel*).

- Mapa *raster* de carreteras e información adicional. Esta segunda textura virtual tiene una resolución de $32\,768 \times 32\,768$ *texels* en el mismo formato que la anterior.

El tamaño de ventana en la ortofotografía es de 2048 *texels* y en el mapa de carreteras de 1024. En una situación óptima sería deseable que ambas

Figura 5.28: Aplicación de gestión y monitorización de información en tiempo real del Centro de Gestión de Tráfico del Noroeste (A Coruña).

fuesen 2048, pero por limitaciones de memoria de vídeo en las máquinas donde se instaló el sistema, se redujo la segunda, puesto que su importancia es menor. En todo caso, la calidad visual resultó adecuada, no apreciándose degradación notable en el mapa de carreteras.

En ambos casos, se utilizó un tamaño de bloque en disco de 512 *texels* y un tamaño de tesela de 128 *texels*. De esta forma se consigue una granularidad bastante fina, tanto para las transferencias y la actualización como para el ajuste del espacio geográfico correspondiente a la ventana cacheada en torno al centro de detalle.

No se utilizó el formato comprimido S3TC DXT1 porque en el momento de la instalación de esta aplicación todavía no estaba implementado en el motor de GeoTextura, pero está previsto añadirlo en una próxima actualización, puesto que reducirá el uso de memoria, disco y transferencias de información, permitiendo usar tamaños de ventana mayores con un coste inferior.

Es necesario que coincidan el tamaño de tesela y el tamaño de *buffer* cuando se utiliza el formato comprimido S3TC DXT1, por limitaciones del sistema gráfico, que no permite hacer correctamente transferencias parciales de un *buffer* a una zona arbitraria de la ventana cacheada de un nivel. En

este caso ambos tamaños se ajustarían a 512 *texels*, y el tamaño de ventana mínimo debería ser 2048, aunque esto último no es un problema puesto que la ocupación de memoria es un 33 % inferior al formato descomprimido con tamaño 1024.

Combinación de las texturas virtuales

En este caso, la combinación de texturas se realiza de forma similar al demostrador genérico, con algunas diferencias que se describen a continuación.

La ortofotografía se toma como base, pero la segunda textura consiste en una imagen *raster* con fondo blanco y no dispone de canal alfa, por lo que no se puede realizar la misma operación que en el caso anterior. En el *fragment shader* se multiplican ambas texturas (sustituyendo por color blanco las que estén desactivadas), puesto que en las pruebas realizadas fue la opción que ofrecía una mayor legibilidad y calidad estética. Las zonas blancas de la textura temática preservan el color de la ortofoto en la multiplicación por lo que el aspecto es similar al que se obtendría con un fundido por canal alfa.

Tras la combinación de las texturas, se realiza un postproceso opcional (el usuario puede activarlo desde la interfaz gráfica) que sombrea las zonas de terreno que tienen peligro de nevada según la cota de nieve suministrada por los servicios de información meteorológica. Utilizando la información de altura del terreno en la posición del *fragment* y la cota de nieve, se funde un color con un nivel de opacidad determinado. Para mejorar el efecto visual, no se hace una transición abrupta en el límite de la zona afectada, sino que se muestra una zona de transición definida por el usuario.

Conclusiones

Las aportaciones del motor GeoTextura a este proyecto han sido el manejo ágil de varias texturas virtuales de gran tamaño y la posibilidad de probar múltiples formas de combinación de dichas capas desde el *shader* para seleccionar la más adecuada.

La flexibilidad de definir la combinación en el *shader* de multitextura posibilita también el sombreado selectivo de la zona con peligro de nieve en función de la altura y la cota de nieve prevista (figura 5.28).

5.5.3. Monitorización de infraestructuras

Otro ejemplo de aplicación ha sido un prototipo desarrollado para la Empresa Municipal de Aguas de La Coruña (EMALCSA), de filosofía similar a la anterior de tráfico, pero orientado hacia la monitorización del estado de los elementos encargados de mantener el suministro de agua (embalses, depósitos, bombas, válvulas, canalizaciones, etc.). En la figura 5.29 se muestran dos pantallas de ejemplo de esta aplicación.

En este caso sí se incluyen datos vectoriales 2D proyectados sobre el terreno, procedentes de sistemas CAD y SIG. Como formato de intercambio para esos datos se ha utilizado DXF, y las entidades de algunas de estas capas contienen estados como grosor y color de línea, por lo que no se pudo realizar la optimización más agresiva descrita en capítulos anteriores.

Las capas más destacables son una selección de información catastral, incluyendo textos con los nombres de las calles y números de portales, y la red completa de abastecimiento de agua, con todas sus acometidas, llaves, etc. Estas capas se componen de millones de vértices y han sido manejadas correctamente por el sistema GeoTextura.

La configuración de capas es similar al demostrador genérico, limitado a una extensión más reducida (A Coruña y área metropolitana), excluyendo la capa de información dinámica, que en este caso no existía como textura, puesto que los datos actualizados en tiempo real están asociados a telemedidas que se representan mediante iconos 3D.

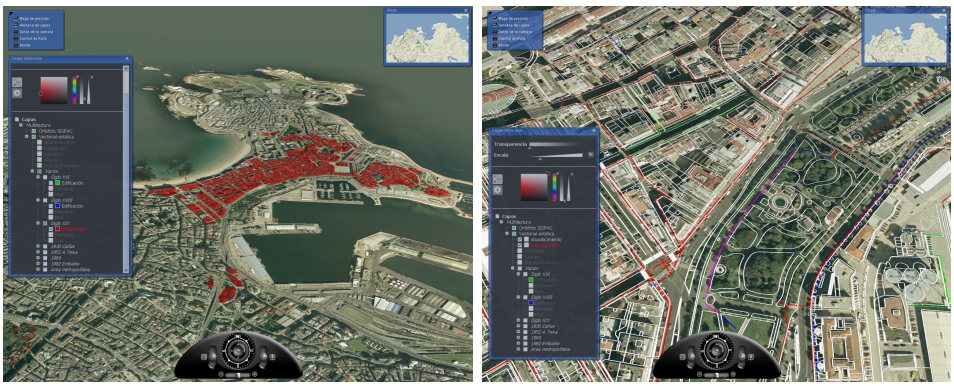


Figura 5.29: Aplicación para la gestión y monitorización de infraestructuras de suministro de agua.

Conclusiones

El proyecto de infraestructuras de suministro de agua ha servido como banco de pruebas para la gestión de capas vectoriales con atributos de color y grosor implícitos, lo cual limita notablemente las posibilidades de optimización, como ya se ha descrito en capítulos anteriores.

5.5.4. Presentación pública de actuaciones locales

Uno de los usos más habituales del sistema de visualización de terreno es la difusión de información sobre actuaciones en un área geográfica determinada. Este es el objetivo de un proyecto en desarrollo para el Ayuntamiento de A Coruña, donde se muestra sobre el terreno la información relativa a las obras realizadas con el Fondo Estatal de Inversión Local (FEIL).

En este caso se ha utilizado como imagen base una secuencia temporal de ortofotografías aéreas del municipio coruñés, correspondiente a los años 2000, 2004, 2006 y 2008, con una resolución de 0,20 m/*texel*. Desde la interfaz de selección de capas se puede cambiar instantáneamente a la fotografía de cualquier año o reproducir la secuencia a modo de vídeo proyectado sobre el terreno (ver figura 5.30).

Figura 5.30: Interfaz para la reproducción de secuencias de texturas virtuales.

Sobre esta imagen base se muestra información vectorial acerca de las obras a realizar, en forma de capas SIG. Para la incorporación de esas capas se han utilizado los formatos SHP y DXF.

Esta aplicación se ha adaptado también para su uso con una interfaz basada en una pantalla multitáctil, la cual se muestra en la figura 5.31.

Conclusiones

El motor GeoTextura ha permitido la combinación de las ortofotografías con la información vectorial y la visión de la secuencia de fotografías o su cambio cuasi-inmediato, lo cual resulta una herramienta muy útil para ver la evolución de la ciudad a lo largo de los años en cualquier zona y con un alto detalle.

Figura 5.31: Aplicación para la presentación de información geográfica mediante interfaz multitáctil.

5.5.5. Análisis y presentación de cobertura radioeléctrica

Otro ejemplo de aplicación incorpora sobre el modelo digital 3D del terreno la información del estado actual y previsto de la infraestructura necesaria para la implantación de la Televisión Digital Terrestre (TDT) en Galicia y las áreas de cobertura correspondientes.

Esta aplicación tiene un doble uso: por una parte sirve como herramienta de trabajo para los técnicos de cara al análisis de la cobertura o la planificación de la ubicación de nuevos dispositivos emisores o reemisores.

Por otra parte sirve de apoyo visual en presentaciones públicas, ruedas de prensa, y reuniones con los responsables municipales para la comunicación eficaz de la información, tanto actual como planificada, de cobertura de TDT en toda la geografía gallega.

Tratándose de un sistema interactivo, es tremendamente eficiente a la hora de atender a consultas o dudas en lugares puntuales, como apoyo a las explicaciones de los técnicos. No importa lo pequeño o aislado que sea el lugar, la aplicación muestra el aspecto real y reconocible mediante la ortofotografía, con su orografía exacta, a cualquier escala que se precise y de forma

inmediata.

Este doble uso técnico/difusión pública es aplicable en muchos proyectos, como el ejemplo anterior de las actuaciones del Ayuntamiento de A Coruña con el FEIL.

Configuración de texturas virtuales

En esta aplicación se ha utilizado una multitextura que combina dos texturas virtuales:

- Ortofotografía comprimida en formato DXT1, equivalente a la del demostrador genérico en tamaño y formato de *texel*. El tamaño de ventana es de 2048 *texels*.

- Información vectorial estática. En esta textura dispone de una serie de capas de información sobre la implantación de la TDT, entre las que se incluyen las siguientes:

  - Equipos instalados (centros emisores, reemisores o *gapfillers*), tanto los actuales como los planificados. Esta capa se compone de entidades de tipo punto, con información asociada sobre cada equipo concreto. Estas entidades se representan tanto en la textura virtual como mediante un icono 3D para mejorar su visibilidad, como se muestra en la figura 5.32. Pulsando sobre el equipo se abre una ficha con la información alfanumérica de dicho equipo y una serie de botones para desplazar la cámara hasta una vista aérea cercana al equipo, una vista subjetiva desde el punto de vista exacto del equipo o para activar las capas de cobertura correspondientes al equipo.
  - Zonas de cobertura asociadas a cada equipo. Se trata de una capa compuesta de entidades de tipo poligonal que se representan aplicándolas de forma semitransparente, de forma que se puede ver la imagen aérea por debajo y la opacidad aumenta cuando en un mismo lugar se solapa la cobertura de varios equipos.
  - Áreas técnicas que identifican las zonas de actuación, codificadas por color.
  - Límites administrativos de municipios, provincias, comarcas, o de las propias áreas técnicas.

Los tamaños de ventana son de 2048 *texels* en ambos casos, y las teselas de 512 y 256 *texels* respectivamente. En el caso de la ortofotografía, el tamaño de tesela corresponde con el tamaño de *buffer* en el origen de datos, por los motivos ya explicados.

Las capas vectoriales han sido optimizadas siguiendo las técnicas descritas en esta tesis doctoral, para permitir su manejo interactivo de forma ágil, cambiando la selección de capas visibles de forma cuasi-instantánea.

Como referencia de la cantidad de información manejada, la capa vectorial con mayor volumen de datos consta de cerca de 1 300 000 entidades poligonales y más de 2 000 000 vértices, distribuidos en más de 400 entidades que se mantienen separadas para poder ser seleccionadas, activadas, desactivadas o coloreadas de forma independiente.

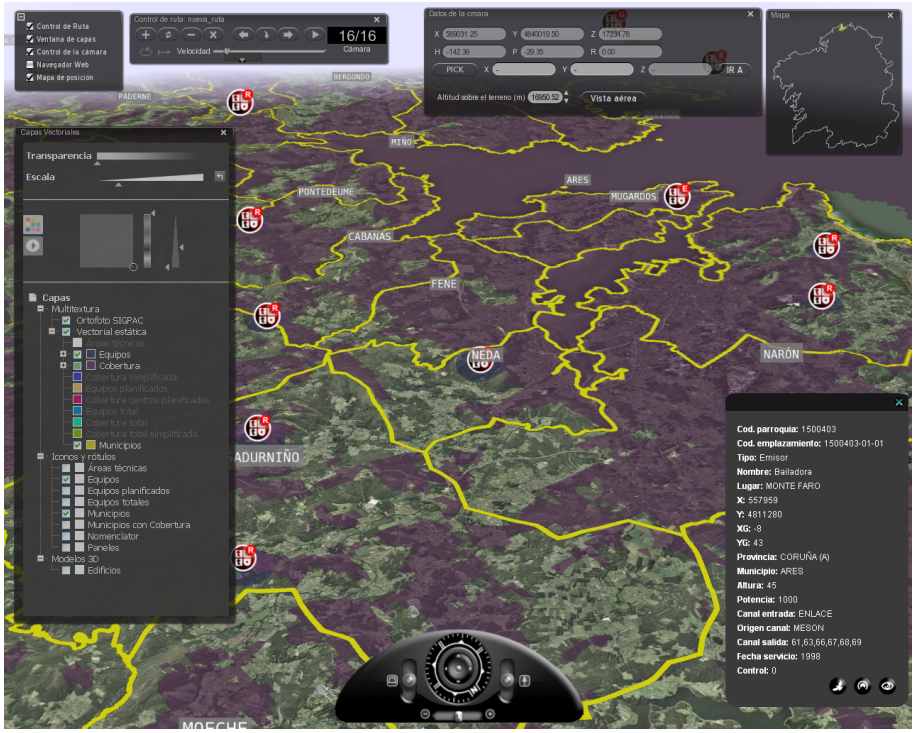


Figura 5.32: Aplicación para el análisis de cobertura de TDT en Galicia.

Conclusiones

El uso del motor GeoTextura en este proyecto ha posibilitado la combinación eficiente de la ortofotografía y las capas vectoriales con gran densidad de información, resultantes de los cálculos de cobertura radioeléctrica. Estas capas, a pesar de su elevado volumen de información, pueden ser activadas, desactivadas de manera cuasi-instantánea, así como variar sus atributos visuales (color, transparencia...) pudiendo acceder independientemente a cada entidad de la capa SIG.

La organización en grafo de escena de los datos vectoriales de estas capas SIG ha permitido también que el usuario pueda seleccionar ágilmente las entidades individuales pulsando con el ratón sobre el terreno 3D y mostrar

información asociada, accediendo a las tablas de la base de datos asociadas a esas capas. Esta facilidad de GeoTextura para manejar independientemente cada entidad y cruzar los datos de sus tablas facilita enormemente las tareas de los técnicos para analizar el estado de la cobertura según los equipos, planificar la ubicación de los nuevos centros a instalar y presentar públicamente esta información de una forma completamente directa y visual, de manera que sea muy fácil de asimilar por la audiencia.

5.5.6. Plan de ordenación del litoral

Otro ejemplo de herramienta para la difusión de información por parte de la administración pública ha sido desarrollado para incorporar sobre el modelo digital de terreno la información del Plan de Ordenación del Litoral (POL) gallego.

La filosofía es similar a la de la aplicación para el análisis y difusión de la información de la TDT, aunque el volumen de información vectorial manejada es un poco superior. La configuración de texturas virtuales es equivalente al caso anterior, pues se trata de una situación prácticamente equivalente (cambiando el tema de los datos vectoriales) en la misma área geográfica.

Las capas vectoriales incluyen información de diverso tipo a lo largo de todo el litoral gallego. La capa con mayor cantidad de información es la de unidades de paisaje, con cerca de 2 500 000 primitivas geométricas sólo en la provincia de A Coruña, distribuidas 32 000 entidades aproximadamente. Estas entidades se han agrupado y se les ha asignado color por tipo de uso de suelo, para facilitar la selección agrupada de todas las entidades de un tipo. En este caso, la interfaz gráfica de selección de capas y entidades sirve también como leyenda del código de colores aplicado por tipo, como se muestra en la figura 5.33, y en ella se tiene acceso a las decenas de tipos de uso del suelo existentes en lugar de las decenas de miles de entidades independientes en la capa original, que por otra parte no tenían otra información asociada, y por tanto se pueden agrupar sin pérdida alguna.

Dado el elevado volumen de información manejado en esta aplicación, no resulta posible tener cargadas en memoria todas las capas simultáneamente. Por este motivo la interfaz de selección de capas permite cargar y descargar capas a criterio del usuario, que puede determinar de esta forma cuáles tener precargadas para poder activar de forma instantánea. Estas cargas se realizan de manera asíncrona, lo cual se indica en la interfaz gráfica con un icono animado que se sustituye por el cuadro de activación y desactivación una vez la carga ha finalizado.

El rendimiento interactivo del sistema se ve afectado puntualmente por la cantidad de información manejada, en concreto en el momento de la activación de las capas de mayor complejidad. Sin embargo, superado este instante inicial correspondiente la activación (que nunca resulta superior a

un segundo), el desplazamiento por el modelo 3D con los datos vectoriales actualizados resulta perfectamente suave. El motivo de este problema es la acumulación de entidades poligonales en las capas inferiores de la pirámide de la textura virtual, y se podría resolver con la implementación de nuevas técnicas de optimización en el optimizador de datos vectoriales para simplificar la geometría en estos niveles.

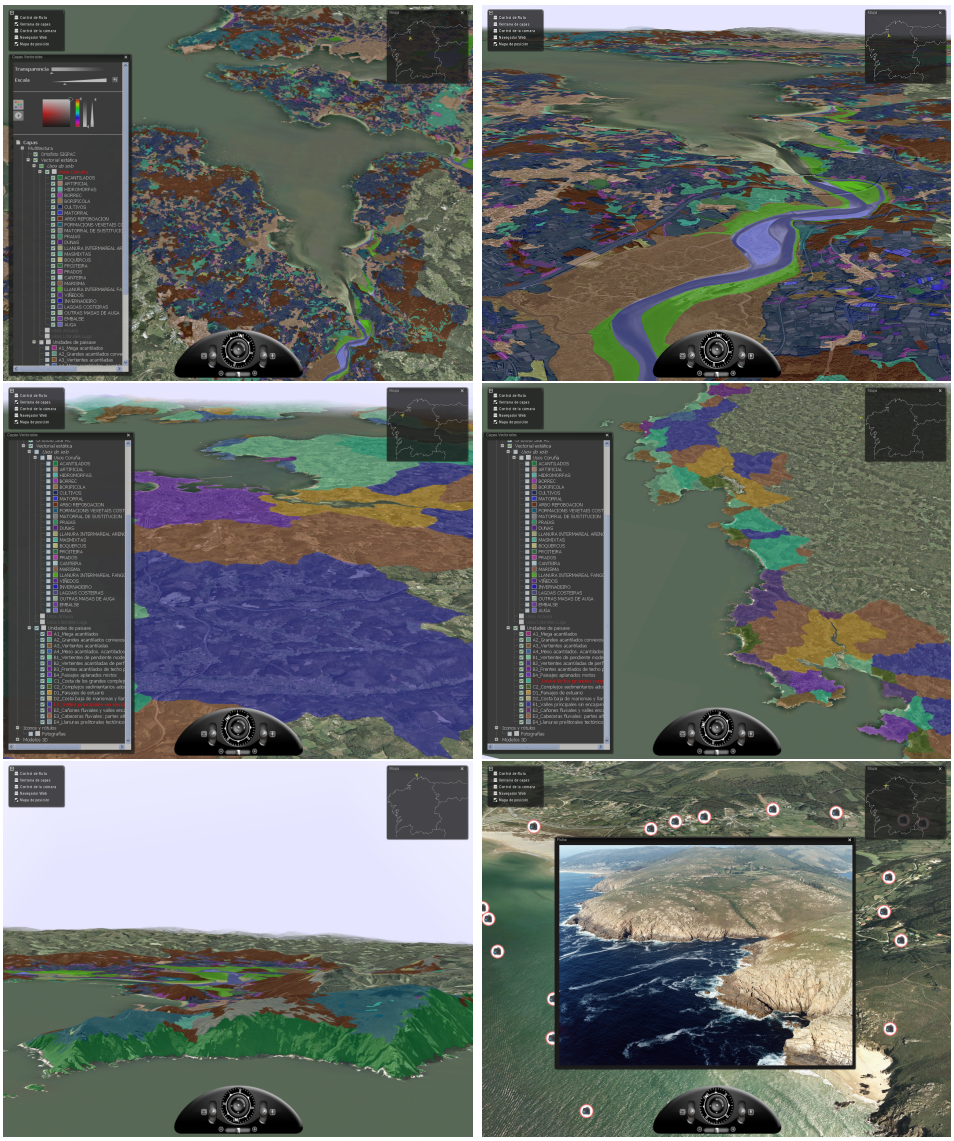


Figura 5.33: Aplicación para la visualización del Plan de Ordenación del Litoral de Galicia.

Conclusiones

En esta aplicación, de forma similar a la anterior de cobertura radioeléctrica, el uso del motor GeoTextura ha posibilitado la combinación eficiente de la ortofotografía con diversas capas capas vectoriales, en este caso extremadamente densas, de forma que ha resultado un banco de pruebas muy bueno para probar el correcto funcionamiento del sistema, especialmente en cuanto a rendimiento.

Adicionalmente, en esta aplicación se han desarrollado unos módulos para la agrupación de entidades por tipo y la asignación automática de colores a las mismas, tarea que de realizarse manualmente sería excesivamente costosa.

La elevada densidad de información vectorial manejada en esta aplicación ha servido para poner a prueba el sistema de optimización y *render* a textura de datos vectoriales, llegando en los casos más extremos a producir leves pausas (inferiores al segundo) en el momento de activación de las capas más complejas. Como ya se ha indicado, esta situación será tema de futuros trabajos en la línea de optimización y generalización cartográfica.

En cualquier caso, el rendimiento, tanto en esta como en las demás aplicaciones descritas, resulta notablemente superior a cualquiera de los otros sistemas de visualización SIG con las mismas posibilidades de visualización y manipulación de los datos, ya sea 2D o 3D, utilizados previamente o evaluados por los usuarios de estas aplicaciones basadas en SANTI y GeoTextura.

Capítulo 6

Conclusiones

Es difícil tomar la decisión de dar por concluido un trabajo de investigación, puesto que cada pequeño paso que se avanza supone abrir una multitud de nuevas líneas de trabajo. Esto está perfectamente reflejado en la famosa cita de Leonardo da Vinci: "*El arte nunca se termina, sólo se abandona*", que podría ser aplicada perfectamente al mundo de la investigación.

Sin embargo, en algún momento hay que tomar la decisión de dar por finalizado el trabajo y recopilar los resultados, que ya han sido analizados en el capítulo anterior. A la vista de esos resultados, podemos afirmar que los objetivos planteados al inicio del trabajo han sido satisfechos en su totalidad y esto se ha demostrado en la práctica con una versión completamente funcional del sistema planteado. Más aún, dicho sistema ha sido implantado en entornos de trabajo reales y está actualmente en uso en diferentes instalaciones, entre las que se puede destacar el Centro de Gestión de Tráfico del Noroeste, perteneciente a la Dirección General de Tráfico, situado en A Coruña.

Durante este trabajo de investigación se ha hecho un estudio del estado del arte de la visualización 3D interactiva de modelos de terreno e información geográfica, partiendo de una visión general de todos los campos involucrados y dirigiéndolo hacia el tema concreto de la visualización de información bidimensional proyectada sobre el modelo geométrico tridimensional del terreno. En concreto se ha hecho especial hincapié en el texturizado de terreno, tomando como referencias fundamentales los trabajos de Cosman[55] y Tanner[144], sobre los cuales se han desarrollado las técnicas descritas en los capítulos anteriores.

En este estudio, se han identificando ciertas carencias en los trabajos existentes, carencias que ya habían sido encontradas en la práctica en varios proyectos desarrollados en el Grupo de Visualización Avanzada en Ingeniería, Arquitectura y Urbanismo (VideaLAB), en los que el autor ha participado [83].

La aproximación tomada para la representación de información vectorial

2D mediante el mapeado de textura, a pesar de recibir críticas en algunas publicaciones ya mencionadas, ha demostrado ser una opción muy adecuada, especialmente para el manejo de grandes cantidades de información vectorial (del orden de millones de vértices). El sistema que se ha desarrollado se adapta perfectamente a cualquier motor de visualización de geometría y solventa los problemas inherentes a una aproximación basada en el *render* de la geometría 3D adaptada al terreno, especialmente en el caso de modelos de terreno que se modifican dinámicamente, por ejemplo con el uso de niveles de detalle contínuos. Por otra parte, las limitaciones al enfoque basado en mapas de texturas planteadas por algunos autores se solucionan completamente con las técnicas propuestas en el presente trabajo.

Este aspecto de no acoplamiento con el motor de geometría es de vital importancia, y ha aportado una solución eficaz a problemas encontrados en anteriores proyectos donde se habían forzado costosos procesos de reteselado y reorganización de la geometría del terreno, con un impacto notable en el rendimiento del *render*.

Los objetivos propuestos en cuanto a volumen de información manejada, rendimiento interactivo, calidad visual y otras prestaciones han sido cumplidos en su totalidad, y los detalles cuantitativos y cualitativos ya han sido descritos en el capítulo anterior. Como limitación se puede resaltar la necesidad de un *hardware* gráfico relativamente reciente (soporte de *Shader Model 4.0* o posterior), lo cual, si bien descarta una parte importante de los ordenadores personales en uso actualmente, no supone un presupuesto elevado para cualquier uso profesional e incluso para ámbitos domésticos.

Un objetivo muy importante que se ha tenido en cuenta durante el desarrollo es la creación de una arquitectura modular, flexible, extensible y versátil, que ha supuesto un excelente marco de trabajo y banco de pruebas para la incorporación progresiva de nuevas características y módulos de conexión a diferentes formatos de información geográfica. El acceso modular a los datos facilita el acceso a través de los servicios y formatos estandarizados definidos por el OGC y otros estándares tanto *de iure* como *de facto*. Esta capacidad se ha demostrado desarrollando diversos cargadores, entre los que se puede destacar el que permite el acceso a fuentes de datos a través del servicio WMS del OGC.

6.1. Futuras líneas de investigación

Como ya se ha mencionado, tras finalizar los trabajos de esta tesis doctoral, y habiendo cumplido todos los objetivos propuestos inicalmente, quedan abiertas múltiples líneas de investigación. A continuación se describen algunas de las más importantes o interesantes.

6.1.1. Optimización de la actualización

De los dos procesos fundamentales del sistema desarrollado: *render* y actualización, el segundo es el más crítico. Las pruebas realizadas con nuevas generaciones de *hardware* gráfico muestran que mientras el rendimiento del *render* aumenta drásticamente, la tasa de transferencia se mantiene aproximadamente igual, puesto que está limitada por el ancho de banda de los buses utilizados.

Por este motivo, una de las líneas más importantes de cara a la mejora del rendimiento del sistema es la optimización de la fase de actualización. El cuello de botella inicial, situado en la caché de segundo nivel ha sido abordado en el diseño de la caché predictiva, descrito en capítulos anteriores, y que en usos normales con *hardware* de gama media alcanza tasas de acierto de caché del 100 %. Por lo tanto, la siguiente línea a estudiar sería la mejora del rendimiento de la actualización síncrona.

Para ello, se está considerando sustituir el mecanismo de carga basado en teselas de tamaño fijo por otro que utilice teselas de tamaño variable en función de la zona que necesita actualización, del tiempo de actualización disponible y de las estimaciones de tiempo de carga según el tamaño de la tesela. Se trata de un cambio significativo que permitiría mejorar la eficiencia de las cargas a costa de una mayor complejidad en la algorítmica de actualización. No es evidente que los beneficios de esta estrategia compensen a sus costes, pero sería interesante analizar ambas alternativas para comparar el rendimiento en diferentes situaciones de uso y en diferentes configuraciones de *hardware*.

Una ventaja adicional de esta carga de teselas de tamaño variable es que permitiría situar la ventana cacheada en TRAM, siguiendo a la posición del centro de interés, con una precisión de *texel* y no de tesela como en la actualidad.

6.1.2. Autoconfiguración

Para un funcionamiento óptimo del motor desarrollado en este trabajo es absolutamente crítica una buena elección de los parámetros de configuración, como tamaño de ventana, de tesela, de bloque en disco, cantidad de memoria dedicada a la caché de segundo nivel, márgenes de seguridad para el control de tiempos de actualización, etc.

El impacto de todos estos parámetros, tanto en rendimiento como en calidad, ha sido estudiado en los capítulos anteriores. Sería de gran utilidad automatizar estas mediciones en el *hardware* utilizado para determinar cuál es la configuración óptima del sistema. En concreto, las pruebas de rendimiento en las cargas de teselas de diferentes tamaños no sólo serviría para determinar el tamaño óptimo sino también para ajustar mejor las predicciones de cara al control del tiempo de actualización.

Este cálculo de los parámetros de configuración óptimos se podría realizar una única vez, durante el proceso de instalación de la aplicación o podría formar parte de la fase de inicialización en cada arranque de dicha aplicación. Cada una de estas dos alternativas tiene sus ventajas e inconvenientes. Por una parte, el hacerlo como parte de la inicialización asegura que esos cálculos siempre estén al día en caso de cambio de *hardware*, pero como contrapartida incrementarán el tiempo de arranque de la aplicación.

Algunas de estas tareas de autoconfiguración podrían incluso realizarse dinámicamente durante la ejecución. Por ejemplo, el tamaño de la caché de segundo nivel podría ajustarse continuamente en función de la RAM disponible en el sistema, de forma similar a cómo lo hace el sistema operativo con la caché de ficheros. Otro ejemplo de ajuste dinámico podría ser el uso de las mediciones de las transferencias del sistema para calcular la estimación del tiempo de carga de teselas y poder ajustar más finamente el control de tiempo.

6.1.3. Manejo de información vectorial

Uno de los puntos del trabajo desarrollado que ofrece mayores posibilidades para continuar la investigación es la gestión de información vectorial. La arquitectura desarrollada delega la visualización de los datos de origen vectorial en ciertos componentes que manejan un grafo de escena o *scene graph* para esta información bidimensional que será representada bajo demanda sobre la textura.

Los trabajos de **generalización cartográfica**, mencionados en capítulos anteriores ofrecen un campo de trabajo amplísimo para complementar el sistema desarrollado. Estos trabajos se han encapsulado dentro de la clase `CargadorRAMSceneGraph`, donde tras acceder al origen de datos, éstos son procesados aplicándoles las operaciones de generalización pertinentes. Estas operaciones pueden ir orientadas hacia dos objetivos: la representación visual y la optimización del rendimiento. En origen, el objetivo de la generalización cartográfica era únicamente el primero de ellos, pero actualmente, en la cartografía digital tiene las dos utilidades.

Representación visual

A pesar de existir numerosos trabajos sobre generalización cartográfica, la mayoría de ellos están orientados a la creación de mapas 2D. El enfoque en este trabajo es claramente diferente, puesto que se trabaja con múltiples escalas (niveles de detalle de la textura) simultáneamente y todas ellas deben encajar perfectamente en las zonas de unión. Por otra parte, estos mapas, a diferencia de los tradicionales no están limitados a leerse orientados hacia el norte.

Estas diferencias en los objetivos y la forma de usar los mapas cartográficos ofrece un campo de investigación muy interesante para mejorar el sistema desarrollado.

Mejora del rendimiento

En las aplicaciones prácticas del motor GeoTextura, descritas al final del capítulo 5, se ha observado que las capas vectoriales con gran cantidad y complejidad de elementos poligonales suponen un cuello de botella en numerosas ocasiones. Esto es debido a la gran acumulación de geometría en las teselas de los niveles inferiores de la textura virtual.

En estos casos se pueden considerar dos alternativas:

- Dividir la generación de las teselas con mayor carga geométrica en varios fotogramas para evitar caídas del rendimiento interactivo.
- Simplificar los contenidos de esas teselas, volviendo de nuevo a técnicas de generalización. Esto se ha realizado en la implementación actual a un nivel bastante elemental, por lo que queda abierto a futuras mejoras.

Un problema encontrado en la optimización de la información vectorial es que se debe buscar un equilibrio entre los aspectos: rendimiento y conservación de la estructura de la información. Dicho de otra forma, si se optimiza el rendimiento del *render* del grafo de escena de los datos vectoriales, se pierde la estructura y agrupación original de los elementos geométricos o entidades del SIG. Si se necesita poder seleccionar individualmente estas entidades para activarlas o desactivarlas, editarlas, cambiar su aspecto, etc. no es posible realizar ciertas operaciones como agregaciones, fusiones o amalgamaciones. Esto limita severamente las posibilidades de optimización y supone un campo interesante para futuras investigaciones.

6.1.4. Mejoras en calidad visual

En cuanto a la calidad visual del sistema, se plantean principalmente dos posibles mejoras.

La más inmediata, y sobre la cual se ha comenzado a trabajar, se aplica a las texturas dinámicas o secuencias temporales. Uno de los aspectos menos agradables visualmente es el cambio brusco de la textura, especialmente en aquellos casos en que en el tiempo del fotograma no se actualiza completamente un nivel de detalle visible. Para solucionar esto se propone el uso de una técnica de "doble *buffer*", utilizando dos cachés de primer nivel (`CacheTRAM`) para la textura virtual, donde mientras se muestra una, se actualiza la otra, para sustituirla una vez actualizada. La diferencia con el "doble *buffer*" clásico es que esta sustitución no se hace de forma instantánea, sino que se realizará un fundido entre ambas imágenes. La duración de este

fundido será de unos pocos fotogramas, configurable por el usuario y limitado siempre por la velocidad de reproducción de la secuencia de imágenes, o lo que es lo mismo, el tiempo de vida de cada imagen de la secuencia.

La otra línea propuesta se plantea a más largo plazo, y sería la mejora de los algoritmos de filtrado de la textura. Actualmente el filtro anisotrópico es el que ofrece mejor calidad de los implementados. Sin embargo, si el aumento de rendimiento de las GPUs continúa como en los últimos años, se podría comenzar a plantear la implementación de algoritmos más complejos como el *eliptic weighted average* descrito por Greene y Heckbert en [81].

Apéndice A

Interfaces del motor de texturizado

A.1. Interfaz genérica. MotorTextura

Las operaciones principales disponibles para cualquier tipo de textura, ofrecidas a través de la interfaz de MotorTextura, se describen a continuación:

init_DRAW Inicialización de la textura. Posibilita que el motor de texturizado correspondiente inicialice todas las estructuras necesarias para su funcionamiento. Se realiza dentro del contexto gráfico. Algunas de las tareas típicas a realizar en este método son la creación de objetos textura de OpenGL y la compilación y enlazado de programas GLSL (*shaders* o programas por vértice y por *fragment*).

guardarEstado_DRAW Guarda en la pila los estados de la máquina OpenGL que el motor de texturizado necesite modificar, para poder posteriormente restaurar el estado de forma que no se afecte accidentalmente al aspecto del resto de la escena. Al delegar esta responsabilidad en la subclase se consigue minimizar el número de estados a guardar y recuperar, puesto que dicha subclase conocerá exactamente cuál es este conjunto mínimo de estados afectados.

restaurarEstado_DRAW Recupera los estados de la máquina OpenGL almacenados cuando se realizó la llamada a guardarEstado_DRAW.

setCentroDeDetalle Notifica al motor de texturizado cuál es la zona de la que se desea disponer del máximo nivel de detalle. Se utiliza una clase abstracta **CentroDeDetalle**, que en la implementación actual es una posición en 2D sobre la superficie del terreno. Este vector se indica en el sistema de coordenadas utilizado, ya sean coordenadas geográficas o cualquier tipo de proyección (habitualmente UTM).

actualiza_DRAW Actualiza la caché en TRAM correspondiente a la textura y realiza cualquier otra operación necesaria para su correcto funcionamiento. Esta llamada se debe realizar una vez por frame para cada textura que necesite actualización. El momento de realizar la llamada puede ser tanto antes como después del *render* de dicho frame, según los requisitos de la aplicación, tal como se describirá más adelante. Para la actualización de las cachés de la textura se tiene en cuenta el CentroDeDetalle, que debería por tanto establecerse anteriormente a la llamada a este método.

Se suministra como parámetro un tiempo máximo que se podrá dedicar a la actualización, y un indicador de si las peticiones de información se realizan de forma síncrona (i.e., los accesos a las fuentes de datos bloquean el sistema hasta que se obtienen dichos datos, por lo que no se puede garantizar que no se supere el tiempo límite indicado) o de forma asíncrona (no se espera por datos que no estén inmediatamente disponibles, y se respeta el tiempo límite asignado a la actualización).

aplica_DRAW Activa la textura (o más correctamente los *shaders* correspondientes) de forma que la geometría dibujada a partir de entonces estará afectada por ella. Este método se encarga de todas las operaciones necesarias. Se puede enviar primitivas geométricas al sistema gráfico sin ninguna restricción.

Los objetos pertenecientes a la clase MotorTextura se construyen indicando un nombre que será utilizado como identificador único. En el momento de la construcción, la textura se añade automáticamente a un registro de objetos MotorTextura que permite recuperar cualquiera de ellos a través de su nombre.

El uso habitual del motor de textura por parte de la aplicación cliente corresponderá a la siguiente secuencia de operaciones:

1. Establecer centro de detalle (y otros parámetros dinámicos, según el tipo de textura).
2. Actualizar cachés del motor de textura.
3. Guardar estados de OpenGL.
4. Aplicar textura(s) virtual(es) dinámica(s).
5. Dibujar geometría afectada por la(s) textura(s) virtual(es) dinámica(s).
6. Restaurar estados de OpenGL.

El orden de las operaciones, no obstante, no tiene por qué ser necesariamente el descrito, sino que puede haber ciertas variaciones por motivos de rendimiento. Este tema se desarrolla en profundidad en el apéndice H.

A.2. Interfaz de textura abstracta. TexturaUnica

TexturaUnica es una clase abstracta que define la interfaz que deben implementar las subclases, que en este caso corresponderán con un sistema de texturizado que puede ser desde un simple objeto textura de OpenGL hasta el complejo sistema de texturas virtuales dinámicas desarrollado en este trabajo (GeoTextura).

Los objetos de la clase TexturaUnica están asociados a una unidad de textura de las disponibles en el sistema gráfico.

El comportamiento de una TexturaUnica será diferente si se utiliza a través de un objeto MultiTextura que cuando se utilice de forma autónoma. Se definen por tanto dos modos de funcionamiento: TEXTURA_UNICA y MULTITEXTURA, que afectarán al comportamiento de las operaciones `init_DRAW`, `guardarEstado_DRAW`, `restaurarEstado_DRAW` y `aplica_DRAW`. La interfaz unificada se respeta, asumiendo por omisión el comportamiento de TEXTURA_UNICA, y se añaden métodos adicionales para su uso desde una MultiTextura.

En cualquiera de los dos casos, la TexturaUnica ofrece a través del método `getFragmentShader` el código GLSL que define una función que permite obtener el valor de la textura para las coordenadas en espacio textura suministradas y correctamente filtrada para el nivel de detalle adecuado. Esta función se denomina `tex`n`_calculaFragment()` donde n es la unidad de texturizado asociada a la textura, puesto que pueden coexistir varias texturas virtuales combinadas dentro del mismo programa GLSL.

A.3. Combinación de texturas y *shaders*. MultiTextura

La clase **MultiTextura** implementa el mecanismo que posibilita la combinación de diversas texturas (a las que se accederá a través de la interfaz TexturaUnica en modo MULTITEXTURA) y la definición de *shaders* (tanto programas por vértice como programas por *fragment*).

Esta clase se utilizará tanto para combinar texturas como para definir *shaders* que utilicen una única textura. A los métodos de la API unificada, se añaden los descritos a continuación:

addTextura Permite añadir objetos de la clase TexturaUnica a la MultiTextura. El propio objeto TexturaUnica ya lleva implícita la unidad de textura que utiliza. Se indica también si la textura añadida estará activa inicialmente.

activaTextura Activa la textura correspondiente a una unidad de texturizado.

desactivaTextura Desactiva la textura correspondiente a una unidad de texturizado.

switchTextura Cambia el estado de una textura (si está activa la desactiva y si está inactiva la activa).

setVertexShader Permite establecer el código GLSL del programa por vértice para la MultiTextura. En caso de no establecer ninguno, se utilizará un programa genérico construido automáticamente a medida de las unidades de textura que se utilicen (figura A.1).

```
void main() {
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_TexCoord[1] = gl_MultiTexCoord1;
    ...
    gl_TexCoord[n] = gl_MultiTexCoordn;
    gl_Position = ftransform();
}
```

Figura A.1: Programa por vértice por omisión para un objeto MultiTextura.

setFragmentShader Permite establecer el código GLSL del programa por *fragment* para la MultiTextura. En caso de no establecer ninguno, se utilizará uno generado automáticamente a medida, al igual que en el caso anterior. El código GLSL de este programa hará referencia a las funciones tex$n$_calculaFragment proporcionadas por los objetos TexturaUnica a través de su método getFragmentShader. En la figura A.2 se muestra un ejemplo de cómo sería este código GLSL.

```
void main() {
    vec4 color = vec4(0.0, 0.0, 0.0, 0.0);
    color += tex0_calculaFragment();
    color += tex1_calculaFragment();
    ...
    color += texn_calculaFragment();
    color /= n+1.0;
    gl_FragColor = vec4(color.xyz,1.0);
}
```

Figura A.2: Programa por *fragment* por omisión para un objeto MultiTextura.

El constructor de un objeto MultiTextura puede utilizar una configuración leída desde un documento XML.

El objeto `MultiTextura` se puede utilizar como interfaz única para las texturas que combina o se puede acceder directamente a los objetos `TexturaUnica`, a criterio del usuario. Por ejemplo, la operación `actualiza_DRAW` de `MultiTextura` distribuirá el tiempo disponible para la actualización entre las diferentes texturas que utilice, en función de ciertos parámetros como prioridad de las texturas, estado de actualización, etc.

A.4. Interfaz de la clase GeoTextura

La clase GeoTextura hereda de TexturaUnica e implementa los métodos comunes a todo motor de texturizado de la interfaz anteriormente descrita. Además, GeoTextura ofrece algunas posibilidades adicionales, como se muestra en el diagrama de la figura A.3.

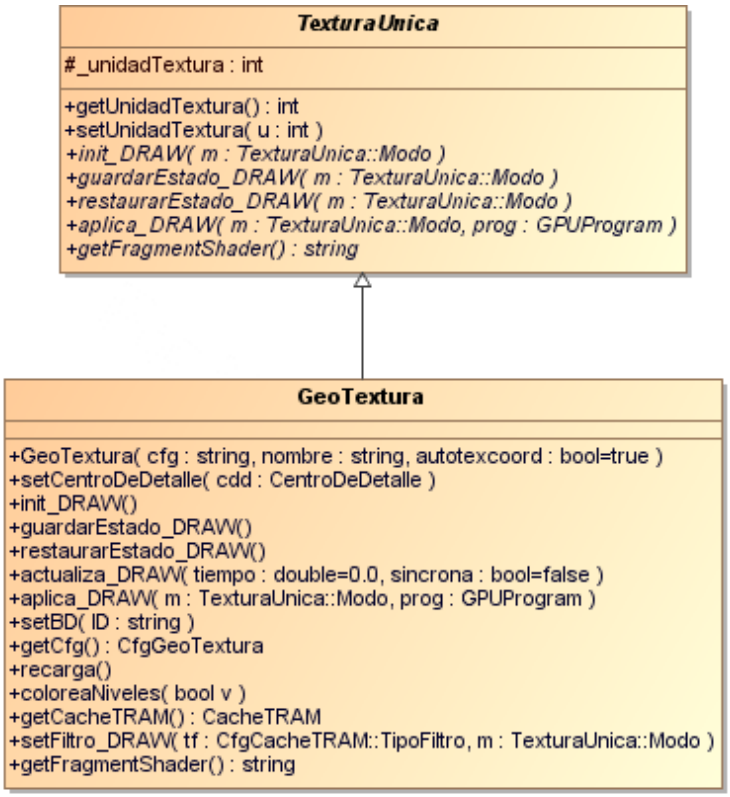


Figura A.3: Diagrama UML de la clase GeoTextura.

setBD Establece una base de datos de entre las disponibles para este motor de textura, identificándola por su nombre. En la configuración de la GeoTextura se identifican las bases de datos que se pueden asignar a cada una.

getCfg Proporciona acceso a la configuración de la GeoTextura. Esta configuración es estática, en el sentido de que se utiliza solamente para la inicialización, no para almacenar estados variables en tiempo de ejecución.

recarga Invalida el contenido de toda la cadena de cachés de forma que se fuerza la recarga completa de los datos desde el origen.

coloreaNiveles Activa un postproceso sobre la textura cargada en las cachés, que aplica un patrón de diferentes colores a cada nivel de detalle de la textura (ver figura 5.1). Esto ha sido de vital importancia durante el desarrollo y especialmente la depuración del sistema, así como para mostrar el funcionamiento del mismo.

getCacheTRAM Proporciona acceso al objeto CacheTRAM que gestiona la GeoTextura.

setFiltro_DRAW Establece el filtro que se aplica sobre la textura. Las opciones disponibles son filtrado por proximidad, bilineal, trilineal y anisotrópico. Los detalles de este proceso se describen en la sección 4.3.4.

A.5. Interfaz de la clase CacheTRAM

La interfaz de CacheTRAM proporciona métodos para operaciones equivalentes a las descritas en la interfaz genérica unificada de MotorTextura, aunque no se hereda de ninguna de las clases de la jerarquía de MotorTextura. GeoTextura delegará en CacheTRAM la mayor parte de sus responsabilidades. Así pues, CacheTRAM tiene métodos para las operaciones típicas como la inicialización, actualización y aplicación, todas ellas dentro del contexto gráfico correspondiente.

Otras operaciones, no dependientes del contexto gráfico, permiten establecer u obtener parámetros como el centro de detalle, configuración de la textura o invalidar los contenidos de la caché. A continuación se describen los métodos fundamentales de cualquier objeto CacheTRAM.

init_DRAW Inicialización del sistema. En esta llamada se ejecutarán las tareas de inicialización que necesitan realizarse dentro del contexto gráfico activo, como la construcción de las texturas necesarias.

actualiza_DRAW Realiza las tareas de actualización periódicas. Típicamente esta llamada se realizará una vez por *frame*, ya sea antes o después del *render*. Los detalles de la actualización se explican en la sección 4.4.

aplica_DRAW Activa la textura para que se pueda aplicar a la geometría que se dibuje a continuación. Se establecen los estados OpenGL necesarios para que el motor de textura funcione correctamente.

getFragmentShader Obtiene el código fuente de la función `calculaFragment`. Esta función en lenguaje GLSL, corazón del motor de textura, se describe en el apartado 4.3.4.

invalida Invalida toda la información contenida en la CacheTRAM. Opcionalmente se puede propagar la invalidación de datos al siguiente nivel de caché (CacheRAM). Este comportamiento se indica en el parámetro `propagar`.

coloreaNiveles Activa la información visual de depuración de los niveles de detalle de la textura (ver figura 5.1). La imagen de la textura tendrá una rejilla superpuesta de un color diferente para cada nivel de detalle. Esta orden se propaga al CargadorTRAM para que los datos que lleguen a través de él vengan marcados con esa rejilla. Tras activar esta opción, es habitual que se fuerce una recarga completa de los contenidos de la caché mediante una llamada al método `invalida`.

getCargadorTRAM Proporciona acceso al objeto CargadorTRAM asociado a la CacheTRAM y encargado de alimentarlo de datos procedentes de la caché secundaria.

getCfg Proporciona acceso a la configuración estática (valores iniciales) del objeto CacheTRAM.

setFiltro_DRAW Establece la técnica de filtrado que se desea utilizar en la textura virtual dinámica. Los tipos de filtro incluyen proximidad, lineal, bilineal y anisotrópico, y su funcionamiento se describe más adelante en la sección de *render* (4.3.4). El cambio de tipo de filtrado supone la reconstrucción de los *shaders* (el código del *shader* se genera en tiempo de ejecución en función de la configuración de la textura), por lo que puede no ser instantáneo.

set/getLODOffset Establece u obtiene el parámetro de desplazamiento que se aplicará al nivel de detalle de textura calculado para cada *fragment*. Este parámetro se describe más adelante en la sección *render* (4.3.4).

set/getMaxAniso Establece u obtiene el parámetro de número máximo de muestras a tomar en el filtrado anisotrópico. El sistema calculará el número de muestras adecuado en función de la posición de la cámara relativa a la superficie del *fragment*, pero se impone un límite máximo a este número de muestras por motivos de rendimiento. Este parámetro

se puede variar sin coste alguno en rendimiento, por lo que sería perfectamente factible establecer un límite diferente en cada fotograma, en función de la carga del proceso de *render*, por ejemplo.

set/getCentroDeDetalle Obtiene o establece la posición del centro de detalle en torno al cual se organizarán los contenidos de la caché y por tanto la información de textura que estará disponible para el proceso de *render*.

setBD Establece el origen de datos al que se accede para obtener la textura. Cada origen de datos puede tener además diferentes configuraciones, de forma que las texturas tengan diferente nivel de detalle, correspondan a áreas geográficas distintas o sus características dinámicas varíen. El cambio de base de datos (u origen de datos) se delega en la clase CacheRAM, que es quien accede directamente a esa fuente a través de CargadorRAM. Los detalles de las bases de datos utilizadas y el proceso de cambio entre ellas serán descritos en los apéndices F y G.

Apéndice B

El proceso de *render* de una GeoTextura

Tal y como se indica en la sección 4.3.4, el proceso de *render* se realiza fundamentalmente en la GPU, mediante un *shader* o programa por *fragment*. Para posibilitar el uso de la textura virtual desde cualquier otro *shader* más complejo o su combinación con otras texturas (no necesariamente virtuales), todas las operaciones se han encapsulado en una función escrita en lenguaje GLSL cuyo código fuente se genera, compila y enlaza en tiempo de ejecución, ya sea durante la fase de inicialización del motor de textura o en una reconfiguración posterior.

Esta función, denominada `calculaFragment`, se encarga de realizar las siguientes tareas:

- Calcular el nivel o los niveles de detalle de textura a los que se necesita acceder.
- Comprobar en la caché la disponibilidad de los *texels* en los niveles de detalle necesarios y en su caso calcular el nivel disponible más adecuado.
- Realizar los accesos necesarios a los *texels* adecuados de la textura 3D (direccionamiento de la caché).
- Combinar los *texels* obtenidos de la textura 3D para realizar el filtrado que corresponda.

Para poder utilizar correctamente el direccionamiento toroidal, la textura 3D que contiene la caché se configura para tener continuidad en los ejes s y t. Esto es así siempre, independientemente del tipo de filtrado que se utilice y la configuración de la textura virtual.

```
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

```
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_CLAMP);
```

La primera tarea a realizar por la función `calculaFragment` es el cálculo del nivel de detalle de la textura que corresponde aplicar al *fragment* que se está procesando.

El cálculo del nivel de textura se basa en el realizado por OpenGL (descrito en las secciones 3.8.8 y 3.8.9, páginas 172 a 179, del documento de especificación de OpenGL 2.1 [135]. Se tomarán estos cálculos como punto de partida para explicar el funcionamiento de la técnica desarrollada.

Un aspecto muy importante a tener en cuenta es que los niveles de detalle de la textura en OpenGL se numeran comenzando por el nivel de máximo detalle, que se considera como nivel 0, aumentando consecutivamente a medida que se reduce la resolución en *texels* a la cuarta parte (la mitad en cada eje) en cada nuevo nivel.

En la técnica desarrollada en este trabajo la numeración se realiza a la inversa que en OpenGL. El nivel 0 corresponderá a la imagen de menor detalle, que será de 1×1 *texels* y los niveles sucesivos se numerarán correlativamente. En el caso de texturas cuadradas, el tamaño del nivel n se puede calcular como $2^n \times 2^n$ *texels*. En caso de texturas rectangulares, el lado mayor será siempre de 2^n *texels*.

Esta diferencia en la numeración facilita conocer el tamaño de la textura a partir de su nivel y no impone ningún límite máximo al tamaño del nivel con detalle más fino. Esto es especialmente importante en el caso de texturas generadas "sobre la marcha" en cuyo caso no existe limitación al detalle que se puede alcanzar, puesto que se genera a partir de información vectorial.

La primera operación realiza el cálculo de la derivada parcial de las coordenadas de textura respecto a las coordenadas en espacio pantalla. Para esto se utilizan las funciones disponibles en la GPU para diferenciar cualquier parámetro respecto a las coordenadas del *fragment* en espacio pantalla.

```
vec2 dx = dFdx( gl_TexCoord[uniTex].st * tex0_tamTex.x );
vec2 dy = dFdy( gl_TexCoord[uniTex].st * tex0_tamTex.y );
```

Estas funciones, que en GLSL se denominan `dFdx` y `dFdy`, no calculan realmente la derivada parcial, que sería un cálculo excesivamente costoso, sino que proporcionan una aproximación más económica en tiempo de cálculo, tal y como se describe en el documento de especificación de GLSL [97] sección 8.8, páginas 65 y 66.

Las variables `dx` y `dy` nos dan una referencia del tamaño de la textura proyectada sobre el *fragment* que se está procesando (espacio pantalla). Esta información servirá para determinar cuál es el nivel de detalle de textura que se debe tomar para aplicar al *fragment*, tal y como se ha detallado en la sección 2.6.1, donde la ecuación 2.5 calcula el factor de escala ρ.

En esa ecuación se toma el máximo de las raices de la suma de los cuadrados de las derivadas parciales de las coordenadas de textura respecto a cada uno de los ejes del espacio pantalla: x e y.

Esto se realizará así para los filtros de proximidad, bilineal y trilineal. En el caso de realizar un filtrado anisotrópico, se calculará de forma distinta, y por ese motivo se guardan los dos términos por separado, en las variables `hx` y `hy`.

```
float hx = sqrt(dx.s*dx.s + dx.t*dx.t);
float hy = sqrt(dy.s*dy.s + dy.t*dy.t);
```

En la figura B.1 se muestran gráficamente los parámetros descritos (`dx`, `dy`, `hx` y `hy`) en un ejemplo de situación anisotrópica que servirá de ejemplo para la explicación del filtrado anisotrópico.

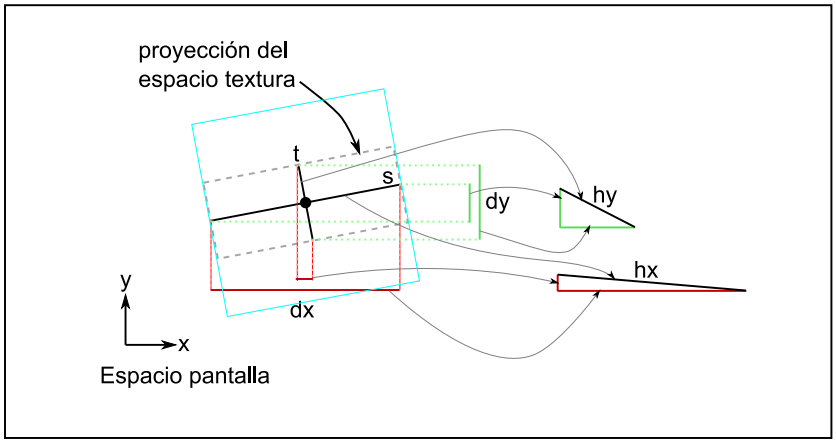


Figura B.1: Proyección de un *texel* del nivel de mayor detalle sobre el espacio pantalla y cálculo de los valores `dx`, `dy`, `hx` y `hy`.

B.1. Filtrado isotrópico

Los filtros de tipo isotrópico soportados por el motor de *render* desarrollado son los siguientes: filtro de proximidad, bilineal y trilineal. En todos ellos, el nivel de detalle que se solicitará a la textura sigue un comportamiento similar al de OpenGL, con las peculiaridades que se describen a continuación.

```
float rho = max(hx, hy);
float lambda = log2(rho);
```

Este valor λ que se ha calculado corresponde al nivel de detalle de la textura según la numeración de OpenGL, que como ya se ha indicado, no es la utilizada en esta técnica. Se convierte a la numeración utilizada por la geotextura restando λ al nivel máximo de la caché.

```
float nivel = MaxNivelPila - lambda;
```

A continuación se ajusta el nivel calculado con el desplazamiento que esté configurado en el sistema. Este valor es equivalente al denominado *LOD offset* en OpenGL y se establece como parámetro de tipo *uniform*, pudiéndose modificar durante la ejecución del sistema. Su misión es permitir al usuario un ajuste fino del nivel de nitidez o suavidad que desea obtener en la textura. Si se aumenta el nivel se obtendrá una mayor nitidez a costa de introducir ruido o *aliasing*, tal y como se describió en la sección 2.6.1. Si se reduce el nivel se eliminará dicho *aliasing*, obteniendo una imagen más suave, pero perdiendo también información, llegando a verse excesivamente borroso si se decrementa demasiado.

```
nivel += tex0_LODOffset;
```

Finalmente se ajusta el nivel calculado al intervalo entre el valor mínimo aceptable (nivel 0) y el máximo disponible en la caché. Este valor máximo se informa al *shader* a través de un parametro *uniform*. El *shader* podría calcularlo a partir de la zona disponible en cada nivel de la pila o pirámide, pero de esta forma se evitan cálculos innecesarios en el proceso más crítico del sistema, que será el *fragment shader*.

```
nivel = clamp( nivel, 0, tex0_maxLOD );
```

Todos los filtros, tanto isotrópicos como anisotrópicos, utilizan unas funciones de apoyo en el *fragment shader*. Estas funciones no se pueden compartir entre diferentes motores de textura porque su código fuente se construye a medida según la configuración de CacheTRAM, y por tanto contendrán el prefijo `texn_` como ya se ha explicado. Antes de describir el funcionamiento de los diferentes filtros que se han implementado en el sistema, se detalla el comportamiento de estas funciones usadas desde `calculaFragment` independientemente del filtrado que se realice.

B.2. Funciones auxiliares

Independientemente del tipo de filtrado que se aplique, la función `calculaFragment` utiliza otras funciones auxiliares para acceder a la información de textura virtual. La primera de estas funciones, y la única que llama directamente es `calculaNivel`.

La función `calculaNivel` recibe dos parámetros: el nivel de detalle de la textura al que se desea acceder y las coordenadas de textura de la posición 2D a la que se desea acceder. El nivel de detalle dependerá de la disponibilidad en la caché, por lo que se puede devolver un nivel inferior al solicitado.

Primeramente se comprueba la disponibilidad de textura cacheada para la zona indicada por las coordenadas de textura. Si la información no está

disponible en el nivel solicitado, se busca secuencialmente en los inferiores (de menor detalle) hasta encontrar un nivel que contenga información válida y actualizada. Esta disponibilidad se calcula con la función auxiliar `estaCubierto`, que se describe más adelante.

```
for ( ; nivel > 0 &&
      !tex0_estaCubierto(nivel, tex_coord) ; nivel-- );
```

Una vez que se conoce el nivel al que se van a solicitar los datos, hay que transformar las coordenadas de textura de espacio de textura virtual al espacio de la textura 3D que almacena la caché. En primer lugar se realiza un ajuste para texturas no cuadradas, en cuyo caso el tamaño en *texels* del nivel de máximo detalle de la pirámide no coincide con uno de los lados de la ventana cacheada.

```
tex_coord.x *= TamPiramideX / tamVentana;
tex_coord.y *= TamPiramideY / tamVentana;
```

Este ajuste se realiza mediante la operación descrita en el código anterior, aunque no se utiliza el código directamente, sino que se precalculan esos factores y se multiplican como una constante sólo en caso de ser distintos de 1. Por ejemplo, para la textura planetaria con doble resolución de ancho que de alto, el código generado sería simplemente el sigiuente

```
tex_coord.y *= 0.5;
```

El siguiente ajuste se realiza sobre las coordenadas de textura para evitar interpolaciones indeseadas en los bordes, que se pueden producir en los niveles de la pirámide en aquellos filtros que activan el modo bilineal de OpenGL para la textura 3D de la caché. Esto ocurre con todos menos con el filtrado por proximidad.

El nivel máximo de la pirámide es una excepción en este cálculo, puesto que puede haber determinados casos en que sí interese esta interpolación. Estos son los casos en que la textura tenga continuidad en los bordes. Por ejemplo, uno de los casos de prueba que se muestran en el capítulo 5 consiste en una textura de todo el planeta en coordenadas geodésicas (longitud en el eje x y latitud en el eje y), de forma que en el eje horizontal la textura tiene continuidad (el extremo este continúa en el extremo oeste) mientras que en el eje vertical no la tiene (evidentemente, el polo norte no coincide con el polo sur).

El sistema permite establecer esta condición independientemente para cada eje, y el código GLSL se generará adaptado a esta configuración, de forma que el *fragment shader* no malgaste tiempo haciendo las comprobaciones. Sin embargo, los casos más habituales son la textura de un área aislada (sin continuidad en ninguno de los ejes) o la textura global del planeta en

coordenadas geodésicas, que tal y como se ha comentado tiene continuidad únicamente en el eje x (longitud).

Así pues, en aquellos casos en que se desee continuidad en la textura no es necesario realizar ninguna operación adicional, pero si en alguno de los ejes no hay continuidad, hay que evitar que en los *texels* del extremo de la textura en ese eje se realice una interpolación bilineal con los *texels* del extremo contrario, produciendo un artefacto indeseado y además bastante notable. La solución que se ha utilizado en estos casos es simplemente desplazar hacia el centro del *texel* del borde las coordenadas de textura que caen en la mitad exterior de ese *texel*.

En la figura B.2 se ilustra esta operación. Se muestra como ejemplo el borde izquierdo de una ventana de la textura.

Puesto que para el correcto funcionamiento del sistema es necesario activar el modo de repetición en los ejes s y t de la textura 3D y en la mayoría de los casos estará activado también el modo de filtrado bilineal de OpenGL, las coordenadas de textura situadas en la zona sombreada en cian provocarán una interpolación bilineal entre los valores de los *texels* del borde de la textura y los de más allá del borde, que corresponderán con el otro extremo de la textura

En el caso de los niveles inferiores de la pirámide, no se ocupa completamente una capa de la textura 3D (ver figura 4.8), por lo que el problema no será la repetición por el otro extremo, sino que los *texels* vecinos del borde corresponden a otra textura de otro nivel o bien no contienen información, y en ningún caso se deberían mezclar. La actuación en este caso es exactamente igual que el anterior, se desplaza esa franja exterior de medio *texel* en el borde de la textura al centro del *texel* del borde en el eje correspondiente.

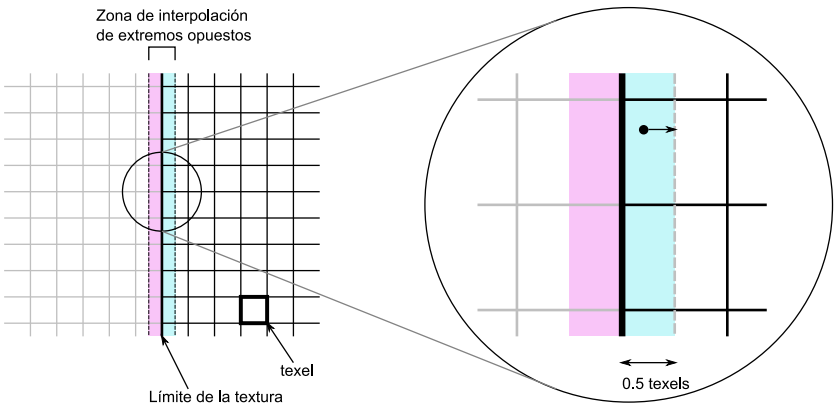


Figura B.2: Ajuste de las coordenadas de textura en los extremos sin continuidad o en la vecindad de otros niveles.

Por ejemplo, el ajuste en caso de la textura planetaria en el nivel de máximo detalle de la pirámide se realizaría mediante el siguiente código GLSL:

```
if ( nivel == MaxNivelPiramide ) {
    vec2 tam_nivel = tex0_getTamTexelsNivel(nivel);

    float ymin = .5 / tam_nivel.y;
    float ymax = 1.0 - ymin;
    tex_coord.y = clamp(tex_coord.y, ymin, ymax);
}
```

La función `getTamTexelsNivel`, referenciada en el código anterior, calcula el tamaño en *texels* de un nivel de la textura virtual. Esta función se utilizará en repetidas ocasiones desde esta y las otras funciones auxiliares.

```
vec2 tex0_getTamTexelsNivel(const in float nivel)
{
    float divisor = round(pow(2.0, MaxNivelPila-nivel));
    return vec2( max(1.0, tex0_tamTex.x / divisor ),
            max(1.0, tex0_tamTex.y / divisor ) );
}
```

En los niveles inferiores de la pirámide, debido a la estructura de la caché almacenada en la textura 3D (ver figura 4.8), no es posible realizar automáticamente esta interpolación en los bordes con continuidad. Sería necesario programarlo en el *shader*, con el consiguiente impacto en el rendimiento (se necesitarían varios accesos a la textura 3D). Puesto que se trata de los niveles más bajos de la textura, y el fallo producido sería de medio *texel* en el borde de unión de la textura, se ha optado por recortar las coordenadas de textura de forma equivalente a como se realiza en el máximo nivel de la pirámide sin continuidad.

```
if ( nivel < MaxNivelPiramide ){
    vec2 tam_nivel = tex0_getTamTexelsNivel(nivel);
    float xmin = .5 / tam_nivel.x;
    float xmax = 1.0 - xmin;
    tex_coord.x = clamp(tex_coord.x, xmin, xmax);
    float ymin = .5 / tam_nivel.y;
    float ymax = 1.0 - ymin;
    tex_coord.y = clamp(tex_coord.y, ymin, ymax);
}
```

Tras los ajustes para evitar problemas en los bordes de la textura, se aplicará un escalado a las coordenadas de textura que las convierte del espacio de la textura virtual al espacio de la textura 3D, dependiendo del nivel o capa en la que se encuentre la información. Las coordenadas de textura coincidirán en ambos espacios en el nivel de máximo detalle de la pirámide, se reducirán a la mitad en cada eje por cada nivel que se descienda y se

aumentarán al doble en cada eje por cada nivel que se ascienda (niveles de la pila).

```
tex_coord /= pow(2.0,MaxNivelPiramide-nivel);
```

En el caso de los niveles inferiores de la pirámide, que no ocupan completamente la capa de la textura 3D (ver figura 4.8), hay que realizar un desplazamiento adicional para ubicar el origen del nivel correspondiente dentro de dicha capa.

```
if ( nivel < MaxNivelPiramide ) {
  for( float i = MaxNivelPiramide - 1.0 ; i > nivel ; i-- )
    tex_coord.x += tex0_getTamTexelsNivel(i).x / tamVentana;
}
```

Finalmente se direccionará la capa de la textura correspondiente al nivel con las coordenadas ya calculadas.

```
float slice = max( nivel - MaxNivelPiramide + 1.5, 0.5 );
return texture3D( tex0_tex, vec3( tex_coord, slice
                                        / ProfTex3D ) );
```

La función `estaCubierto` recibe los mismos parámetros que `calculaNivel` y devuelve un valor lógico indicando si el *texel* correspondiente a las coordenadas de textura para ese nivel está disponible en la caché. Este cálculo lo realiza a partir de la información de cobertura de niveles de la caché, suministrada mediante parámetros *uniform* que indican los límites del área rectangular disponible en cada nivel con datos válidos y actualizados.

Al igual que sucedía con los extremos de los niveles de la pirámide, tal y como se describió anteriormente, en los extremos de los niveles de la pila se producen errores debidos a la interpolación bilineal de *texels* que no son contiguos. En este caso, nunca existirá contiguidad, puesto que los niveles de la pila, por definición, contienen un subconjunto propio del nivel completo de la textura virtual. Es decir, hablamos siempre de niveles incompletos por lo que no habrá continuidad en los extremos.

Por este motivo, se realiza una comprobación en la función `estaCubierto` que evita el uso del *texel* exterior de la ventana cacheada. En la comprobación de si un *texel* está cubierto por el área disponible en un nivel determinado se ajustará previamente dicha área con un margen de seguridad de un *texel* de ancho.

```
vec2 tam_texels = tex0_getTamTexelsNivel(nivel);
vec2 margen_seguridad = 1.0 / tam_texels;
```

En la figura B.3 se puede ver el efecto de la interpolación (con una zona ampliada en la imagen inferior para apreciar el detalle). Por una parte se aprecia claramente que la zona afectada es mínima, por lo que la pérdida de ese *texel* (que se sutituirá por el del nivel inmediatamente inferior) no supone ningún problema. Por otra parte se demuestra que a pesar de su reducido tamaño, esa interpolación produce un artefacto claramente visible y que debe ser eliminado.

Para mostrar mejor el efecto, en la figura se ha utilizado un tamaño de ventana reducido (512×512 *texels*), notablemente inferior a la resolución de la pantalla (1280 pixels de ancho). En el uso habitual con una ventana superior, la zona del borde quedaría generalmente fuera de la zona visible, salvo casos en que la caché no se actualice lo suficientemente rápido.

El área cubierta se obtiene del parametro *uniform* `limites`, que es una tabla de vectores de 4 elementos (un vector por cada nivel de la textura) con las coordenadas que definen el área rectangular disponible en ese nivel con datos válidos y actualizados. Estas coordenadas se representan en espacio textura.

```
#define MIN_X tex0_limites[int(nivel)].x
#define MIN_Y tex0_limites[int(nivel)].y
#define MAX_X tex0_limites[int(nivel)].z
#define MAX_Y tex0_limites[int(nivel)].w
```

La comprobación de estos límites depende de nuevo de la continuidad del espacio de la textura virtual en los bordes. Si en un eje no hay repetición, se realizan las siguientes comparaciones, teniendo en cuenta el margen de seguridad anteriormente mencionado. Por ejemplo, si en el eje t no hay repetición, el código quedaría como se indica a continuación.

```
if (
    tex_coord.t - margen_seguridad.y < MIN_Y
    ||
    tex_coord.t + margen_seguridad.y > MAX_Y
   )
    return false;
```

En cambio en los ejes en que sí haya continuidad, se puede dar el caso de que el valor mínimo de los límites (una vez ajustado al espacio real de la textura entre 0 y 1) sea superior al máximo. Esto sucederá cuando la ventana cacheada cruce el límite de la textura en dicho eje, de forma que se divide entre los dos extremos de la textura. Si no hay repetición, la fase de actualización, descrita en el apartado 4.4, no permitirá que se produzca tal situación.

Siguiendo con el ejemplo de la textura planetaria, con repetición en el eje s, las comprobaciones a realizar en este caso quedarían de la siguiente manera.

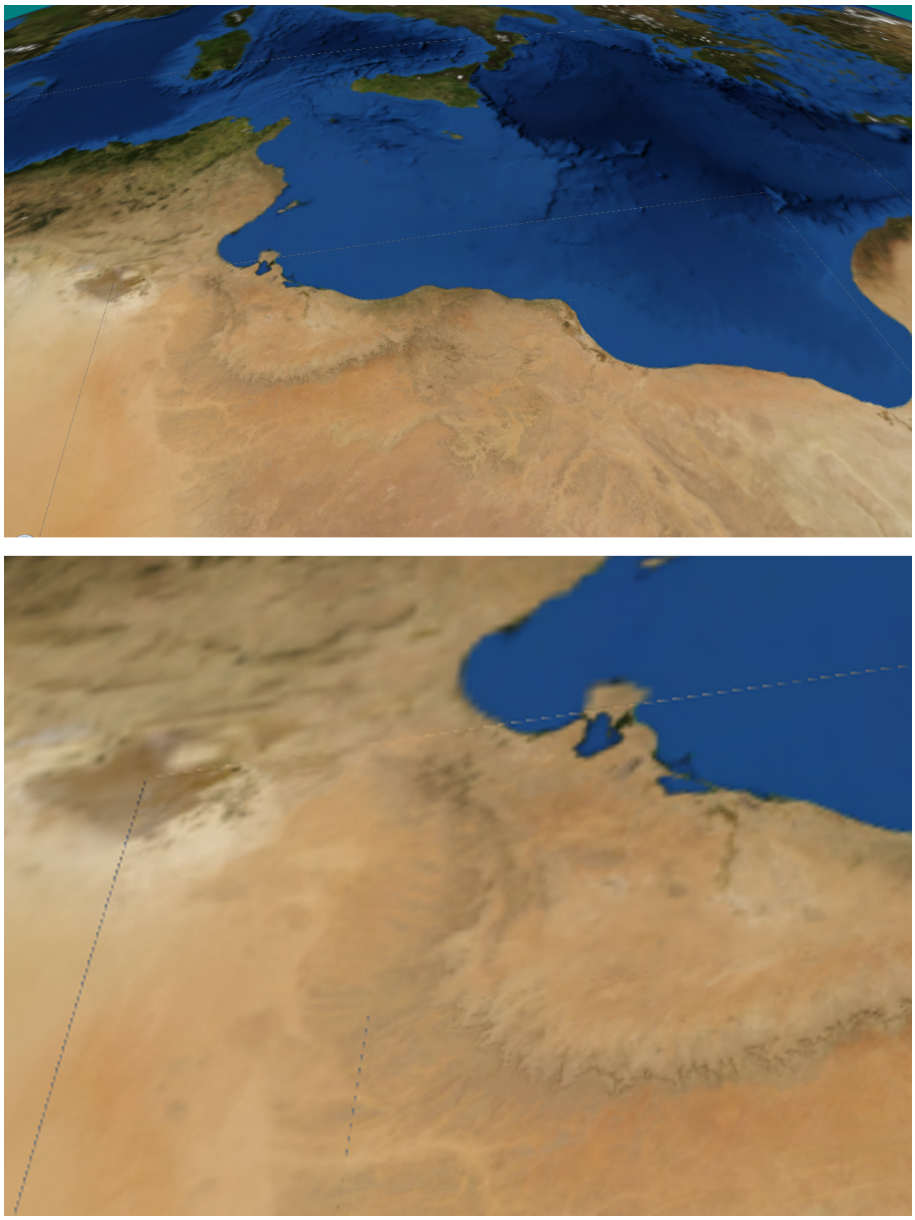


Figura B.3: Defecto visual por interpolación en los bordes de las ventanas de caché de los niveles de la pila.

```
if ( MAX_X <= 1.0 ) {
    if (
        tex_coord.s - margen_seguridad.x < MIN_X
        ||
        tex_coord.s + margen_seguridad.x > MAX_X
       )
```

```
            return false;
    }
    else {
        if (
            tex_coord.s - margen_seguridad.x < MIN_X
            &&
            tex_coord.s + margen_seguridad.x > (MAX_X - 1.0)
           )
            return false;
    }
```

Si no se cumple ninguna de las condiciones indicadas en el código anterior, se asume que las coordenadas de textura suministradas están disponibles en el nivel indicado y se devuelve un valor lógico verdadero como resultado de la función.

B.3. Filtrado por proximidad

El filtro por proximidad activa ese mismo tipo de filtrado (`GL_NEAREST`) en OpenGL para la textura 3D que contiene la caché, tanto para la reducción como para el aumento de la textura. De esta forma se obtendrá siempre el valor del *texel* más cercano a las coordenadas de textura indicadas.

```
glTexParameteri(GL_TEXTURE_3D,
                GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_3D,
                GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

El nivel de detalle calculado de la forma descrita anteriormente para los filtros isotrópicos se redondea al entero más próximo, de forma que se realiza un único acceso a la textura 3D.

```
return tex0_calculaNivel(round(nivel), tex_coord);
```

El filtrado por proximidad, aunque es el más barato en tiempo de cálculo, no se suele utilizar, puesto que ofrece un aspecto muy ruidoso, especialmente al aproximarse la imagen tiene un aspecto muy pixelado (ver figura B.4).

B.4. Filtrado bilineal

En el filtro bilineal (de hecho en todos a excepción del de proximidad descrito en el apartado anterior) la textura 3D se configura para acceder a ella con el filtro bilineal de OpenGL activado, tanto para el aumento como para la reducción de la textura.

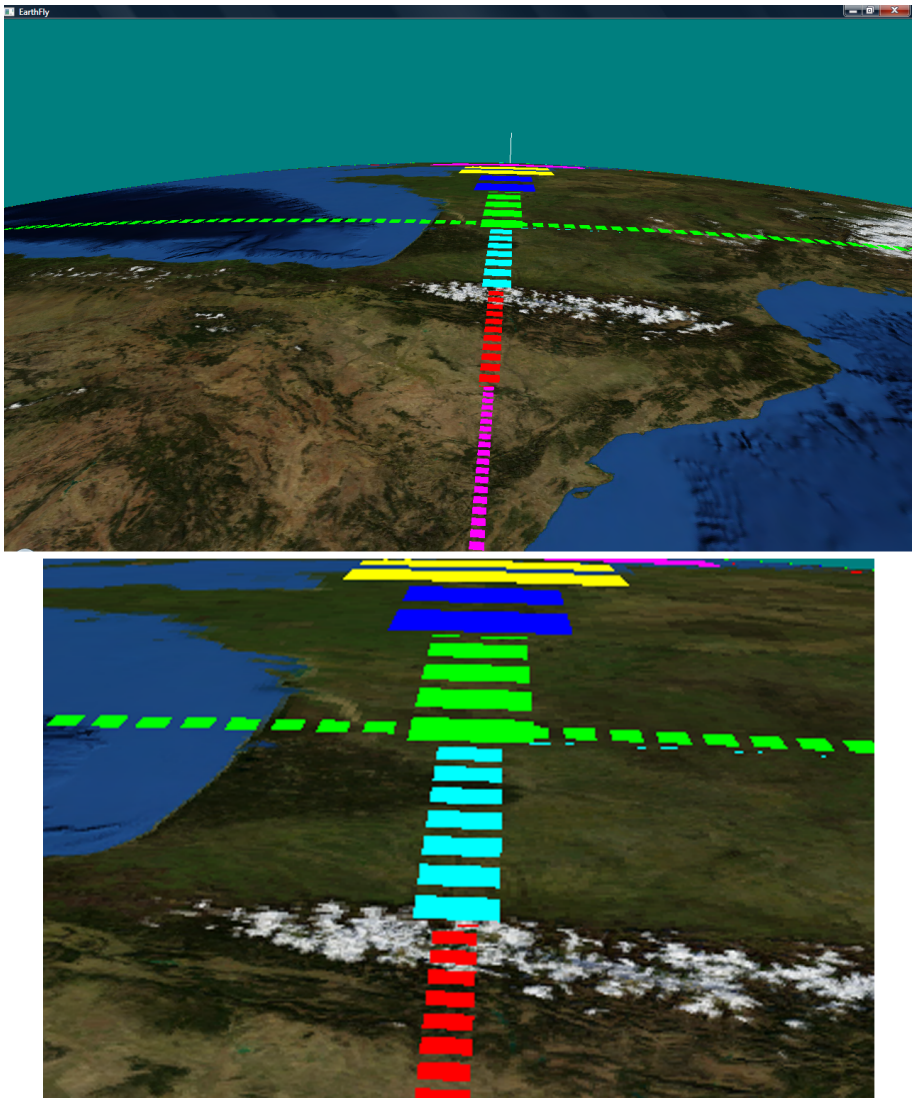


Figura B.4: Filtrado por proximidad con los niveles marcados con un código de colores.

```
glTexParameteri(GL_TEXTURE_3D,
                GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_3D,
                GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

Igualmente, en este caso se redondea el nivel de detalle calculado al entero más próximo y se accede a la capa correspondiente de la textura 3D.

```
return tex0_calculaNivel(round(nivel), tex_coord);
```

El filtrado bilineal suaviza el aspecto pixelado del anterior (ver figura B.5), pero los cambios entre niveles de la pirámide son bruscos, tanto en la animación al acercarse o alejarse del objeto texturizado, como en una imagen estática, donde se pueden apreciar las lineas divisorias entre niveles (ver figura B.5).

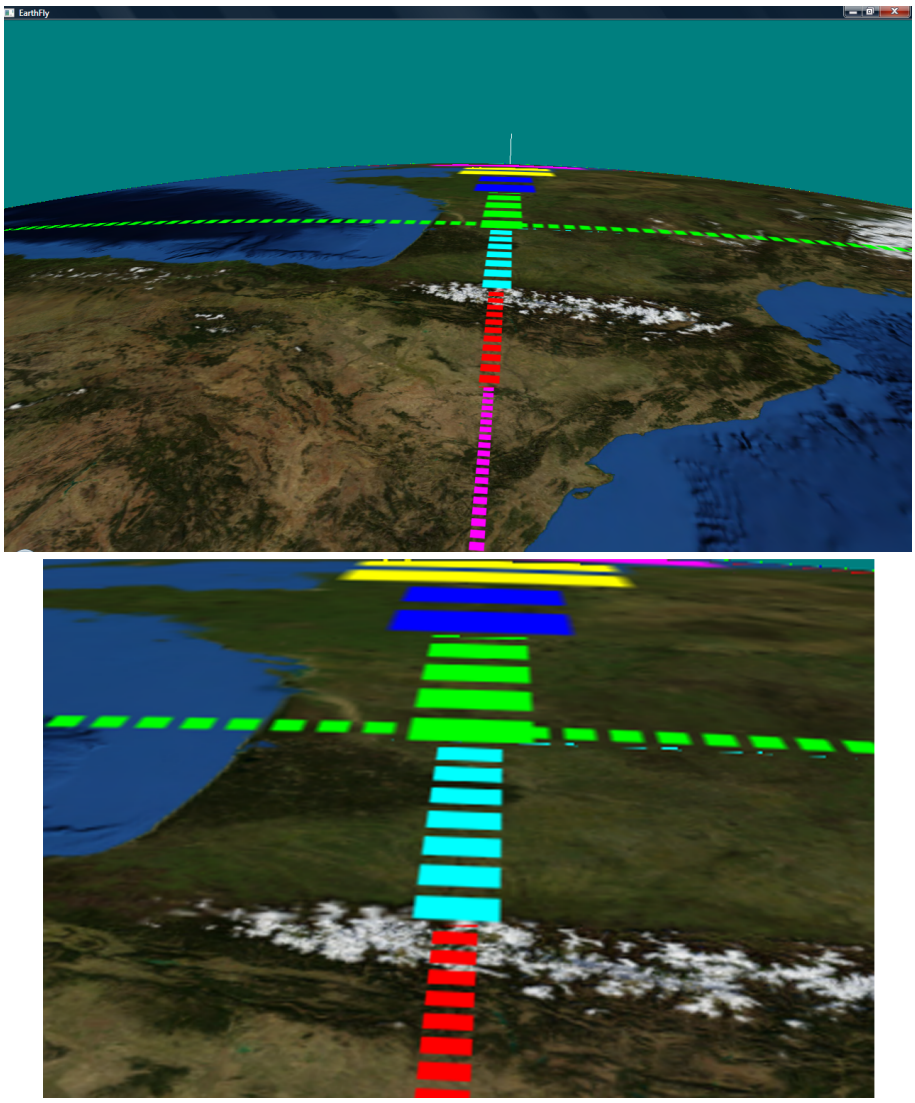


Figura B.5: Filtrado bilineal con los niveles marcados con un código de colores.

B.5. Filtrado trilineal

A diferencia de los dos anteriores, el filtrado trilineal sí tiene en cuenta el valor real del nivel de detalle calculado, que no coincide exactamente con uno

de los niveles prefiltrados de la pirámide a no ser que coincida en un número entero. Lo habitual es que esté situado entre dos niveles de la pirámide, en cuyo caso se tomarán muestras (filtradas bilinealmente) en esos dos niveles circundantes y se interpolará linealmente entre ellas. De esta forma se tiene en cuenta la distancia a ambos niveles enteros, de forma que tenga más peso la imagen que está más cerca del nivel calculado.

```
// Niveles enteros para mezclar/interpolar
float niv_sup = ceil(nivel);
float niv_inf = floor(nivel);
float a = niv_sup - nivel;

// Interpolamos entre los dos niveles de la
// textura prefiltrados (GL_LINEAR)
return mix( tex0_calculaNivel( niv_sup, tex_coord ),
            tex0_calculaNivel( niv_inf, tex_coord ), a );
```

Este filtro soluciona el problema de los cambios entre niveles de detalle, que resultan absolutamente suaves. La transición funde los niveles de forma que no es apreciable, tanto en animación como en imágenes estáticas (ver figura B.6).

Volviendo al ejemplo de la figura B.1, el filtro trilineal tomaría el mayor de los valores (`hx` en este caso) para calcular el nivel de detalle de la textura y muestrearía la zona representada en el cuadrado de color cian, por lo que se añadiría más información de la que corresponde a esa zona, de ahí que se produzca el efecto de desenfoque excesivo en el eje t de la textura, que se intentará solucionar o al menos reducir con el uso del filtro anisotrópico.

B.6. Filtrado anisotrópico

El filtrado anisotrópico soluciona los problemas de excesiva borrosidad del filtrado trilineal (y por supuesto también de los otros filtros isotrópicos) en situaciones en que la superficie texturizada se ve con un ángulo muy oblicuo. Su base teórica ha sido descrito ya en la sección 2.6.1, por lo que no se repetirá aquí. Se describirá directamente la implementación que se ha realizado.

En primer lugar, para la elección del nivel de detalle que se va a utilizar, se toman los valores `hx` y `hy` descritos anteriormente y utilizados en los otros filtros (ver figura B.1). En lugar de calcular el factor de escala (ρ) como el máximo de ambos valores y tomar una única muestra, se tomará un valor comprendido dentro del intervalo [`hx`,`hy`] (idealmente el mínimo) y en ese nivel se tomarán varias muestras a lo largo de la **línea de anisotropía** (que coincidirá con uno de los ejes en espacio textura) para aproximar el contenido real de la textura que coincide con el *fragment* procesado.

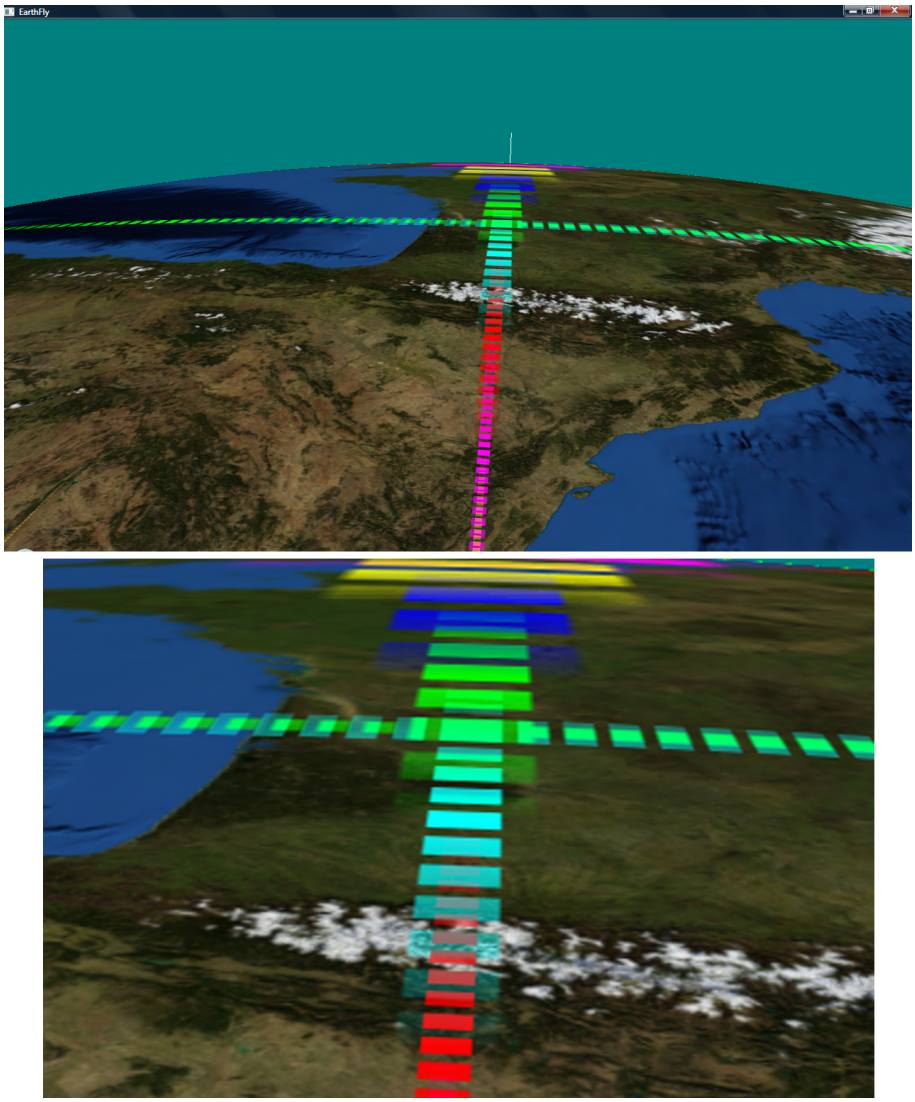


Figura B.6: Filtrado trilineal con los niveles marcados con un código de colores.

La relación entre ambos valores determina la **ratio de anisotropía**. Esta ratio determinará el número de muestras trilineales que se van a combinar para calcular el valor final del *fragment*.

Sin embargo, no se toma directamente la ratio calculada, puesto que en angulos de visión muy cercanos al plano texturizado el número de muestras se hace muy elevado. Cada muestra, debido a que se le aplica un filtrado trilineal, supone una cierta cantidad de operaciones aritméticas, pero lo más importante son los ocho accesos a textura que realiza. Estos accesos a textura son una de las operaciones más costosas que realiza el *fragment shader*, con un impacto directo en su rendimiento. Además, en el caso del sistema de

GeoTextura, se añaden las tareas adicionales de direccionamiento de la caché que se han descrito en esta sección.

Por estos motivos, se establece un límite máximo al número de muestras trilineales que se tomarán para combinarlas en el filtrado anisotrópico. Este límite se proporciona al *shader* mediante un parámetro de tipo *uniform*, de forma que se puede variar en cada fotograma sin afectar al rendimiento, permitiendo un ajuste dinámico de la calidad del filtrado en función del estado de carga del sistema de *render*.

```
float ratio_anisotropia = max(hx,hy) / min(hx,hy);
float num_muestras = min(ceil(ratio_anisotropia), MAX_ANISO);
float rho = max(hx, hy) / num_muestras;
```

Una vez calculado el factor de escala (ρ), el nivel que finalmente se accederá se calcula y se ajusta como se ha descrito para los filtros isotrópicos.

```
float lambda = log2(rho);
float nivel = MaxNivelPila - lambda;
nivel += tex0_LODOffset;
nivel = clamp( nivel, 0, tex0_maxLOD );
```

Llegados a este punto, se toma el mayor de los dos valores `hx` y `hy` que corresponderá con la línea de anisotropía. Se pasa también de trabajar en *texels* al espacio de coordenadas de textura, dividiendo por el número de *texels* de la textura completa. La variable `dif_st` contendrá la magnitud en coordenadas de textura del *fragment* a lo largo de la línea de anisotropía.

```
vec2 dif_st;
if ( hx > hy)
    dif_st = dx / tex0_tamTex;
else
    dif_st = dy / tex0_tamTex;
```

Se tomarán `num_muestras` a lo largo de un segmento de la línea de anisotropía, centrado en las coordenadas de textura asociadas al *fragment* y con el tamaño en espacio textura que se ha calculado en `dif_st`. Estas muestras se promediarán en el valor que retorna la función `calculaFragment`.

```
int num_muestras_int = int( num_muestras );
vec2 dif_st_muestra = dif_st / (num_muestras+1.0);
vec2 tex_coord = gl_TexCoord[uniTex].st -
                    dif_st_muestra * (num_muestras / 2.0);
vec4 color_fragment = vec4(0.0, 0.0, 0.0, 0.0);
for( int i=0 ; i<num_muestras_int ; i++ ){
    color_fragment += mix(
```

```
                tex0_calculaNivel(niv_sup, tex_coord),
                tex0_calculaNivel(niv_inf, tex_coord), a);
        tex_coord += dif_st_muestra;
    }
    return color_fragment / num_muestras;
```

En el ejemplo de la figura B.7 resultarían tres muestras ubicadas como se indica. La aproximación no es exacta, en este caso se pierde un poco de información en el eje t, pero el resultado visual es notablemente mejor que en los filtros isotrópicos (ver figuras B.9 y B.10). El error cometido se puede llevar hacia una mayor nitidez o una mayor suavidad ajustando el desplazamiento del nivel de detalle (`LODOffset`) tal y como ya se ha explicado.

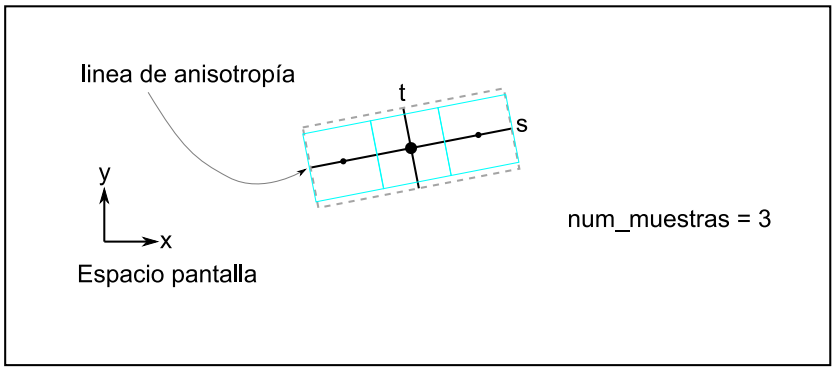


Figura B.7: Ejemplo de filtrado anisotrópico mediante tres muestras trilineales.

Las figuras B.9 y B.10 muestran una comparativa de la calidad visual en una imagen estática mediante los diferentes filtros implementados tal y como se han descrito. En el capítulo 5 se profundiza en la comparativa entre estas técnicas, tanto en calidad visual como en rendimiento.

B.7. Parámetros utilizados por el *shader*

La función `calculaFragment` recibe una serie de parámetros relativos a la configuración de la textura virtual y a su caché. Estos parámetros se suministran mediante dos vías principalmente. En el caso de parámetros que varíen o puedan variar durante la ejecución, se utilizarán parámetros de tipo *uniform* de GLSL. En caso contrario, se declararán como símbolos o "macros" del precompilador, de forma que se utilizarán como una constante dentro del código. Cualquier variación a los parámetros de este último tipo supone regenerar el código GLSL de los *shaders*, recompilarlos y reenlazarlos, por lo que no se deberían modificar salvo casos de fuerza mayor. Además de parámetros de configuración, algunos valores calculados de uso repetido en el código o de cálculo costoso se pasarán como macros para evitar cálculos innecesarios a la GPU durante la ejecución de los *shaders*.

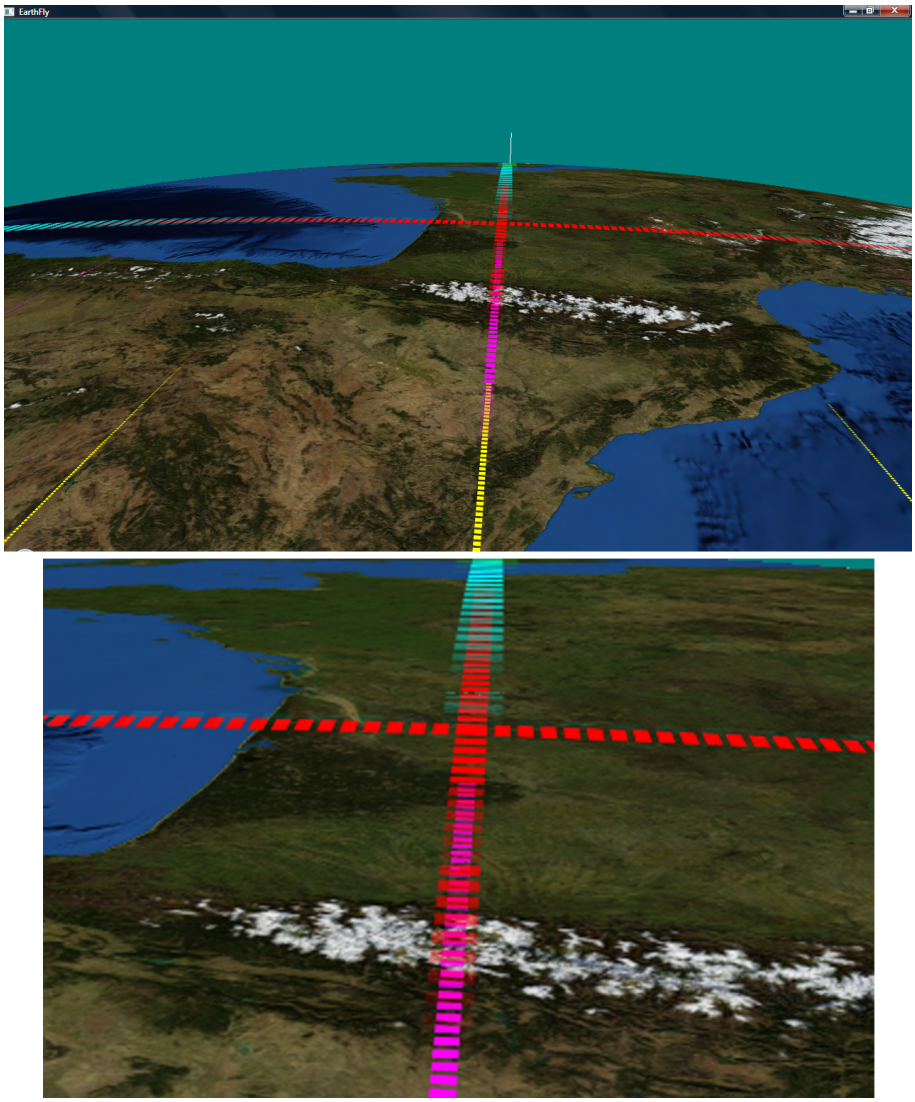


Figura B.8: Filtrado anisotrópico con los niveles marcados con un código de colores.

A continuación se describen los parámetros utilizados.

`uniTex` (macro) Unidad de texturizado a la que está asociada la GeoTextura. El motivo de utilizar una macro del precompilador en lugar de un parámetro es que en la arquitectura utilizada en el desarrollo de este trabajo (NVIDIA serie 8) no se permite direccionar las coordenadas de textura mediante una variable.

`sampler3D texn_tex` (*uniform*) Textura 3D que contiene la caché de la textura virtual. Es la única textura a la que se accederá desde la función `calculaFragment`.

`vec2 texn_tamTex` (*uniform*) Tamaño de la textura virtual (número de *texels* en el nivel de máximo detalle).

`tamVentana` (macro) Tamaño de lado de la ventana cacheada. Se asume una ventana cuadrada. Este tamaño determina el ancho y alto de la textura 3D y por tanto también de cada nivel de la pila.

`float texn_maxLOD` (*uniform*) Nivel de detalle máximo con información útil. Este parámetro acelera el trabajo del *fragment shader*, puesto que evita comprobaciones en los niveles de detalle que ya se sabe a priori que no contienen información válida.

`float texn_LODOffset` (*uniform*) Desplazamiento que se aplica al nivel de detalle calculado para el *fragment*. Permite hacer un ajuste fino para conseguir mayor nitidez o mayor suavidad en la textura.

`float texn_maxAniso` (*uniform*) Límite máximo al número de muestras trilineales que se combinarán en el filtro anisotrópico. Este parámetro sólo existirá en el caso de utilizar filtrado anisotrópico.

`vec4 texn_limites[NumNiveles]` (*uniform*) Límite del área rectangular disponible en caché para cada nivel. Esta información es la que utiliza la función `estaCubierto` para determinar si hay información válida para unas coordenadas de textura concretas.

`MaxNivelPiramide` (macro) Máximo nivel de la pirámide.

`MaxNivelPila` (macro) Máximo nivel de la pila.

`MinNivelPila` (macro) Mínimo nivel de la pila (uno más que el máximo de la pirámide).

`ProfTex3D` (macro) Profundidad (número de capas) de la textura 3D (número de niveles de la pila más las capas utilizadas para la pirámide).

B.8. Comparación entre OpenGL y GeoTextura

Los filtros implementados por la textura virtual dinámica GeoTextura corresponden en cierta manera con los que realiza OpenGL para las texturas ordinarias. A continuación se describen similitudes y diferencias entre unos y otros.

OpenGL diferencia dos situaciones a la hora de aplicar los filtros: el aumento y la reducción de la textura.

En el caso de aumento, permite tomar la muestra más cercana (`GL_NEAREST`) o interpolar los valores de las cuatro muestras que rodean la coordenada de textura (`GL_LINEAR`).

GeoTextura	OpenGL (aumento)	OpenGL (reducción)
Proximidad	`GL_NEAREST`	`GL_NEAREST_MIPMAP_NEAREST`
Bilineal	`GL_LINEAR`	`GL_LINEAR_MIPMAP_NEAREST`
Trilineal	`GL_LINEAR`	`GL_LINEAR_MIPMAP_LINEAR`
Anisotrópico	`GL_LINEAR`*	`GL_LINEAR_MIPMAP_LINEAR`*

Cuadro B.1: Correspondencia entre los filtros de GeoTextura y los de OpenGL.

En el caso de reducción, se pueden usar los anteriores de la misma manera, en cuyo caso se aplican a la textura en su nivel de mayor detalle o bien se pueden utilizar mipmaps, realizando una interpolación por proximidad o lineal tanto en el espacio de la textura (s, t) como en el nivel de detalle (λ).

A diferencia de las texturas de OpenGL, en el caso de GeoTextura, la textura virtual no está completa en TRAM, por lo que no se puede prescindir de los niveles de detalle a la hora de interpolar. Así pues, en los casos de aumento, se tomará el valor de máximo detalle disponible en la caché para la zona mapeada, que no necesariamente será el máximo detalle de la textura virtual, aunque en condiciones habituales (recursos suficientes para la actualización de las cachés a la velocidad que se avanza, cálculo correcto del centro de detalle y elección adecuada del tamaño de ventana) sí lo será.

En los casos de reducción de la textura, el funcionamiento sería similar a uno de los modos que utilizan mipmaps, pero con la particularidad de que no siempre está disponible el nivel que resulta de los cálculos definidos en las especificaciones de OpenGL, por lo que el nivel utilizado puede ser inferior al previsto.

En caso de que la caché se actualice correctamente, el centro de detalle se sitúe de forma adecuada y el tamaño de ventana sea suficiente, la equivalencia entre los modos de filtrado de GeoTextura y los de OpenGL se detallan en la tabla B.1.

El resultado del filtrado anisotrópico puede ser similar al disponible por *hardware* en las GPUs actuales mediante la extensión `EXT_texture_filter_anisotropic` para el filtrado anisotrópico de texturas[54].

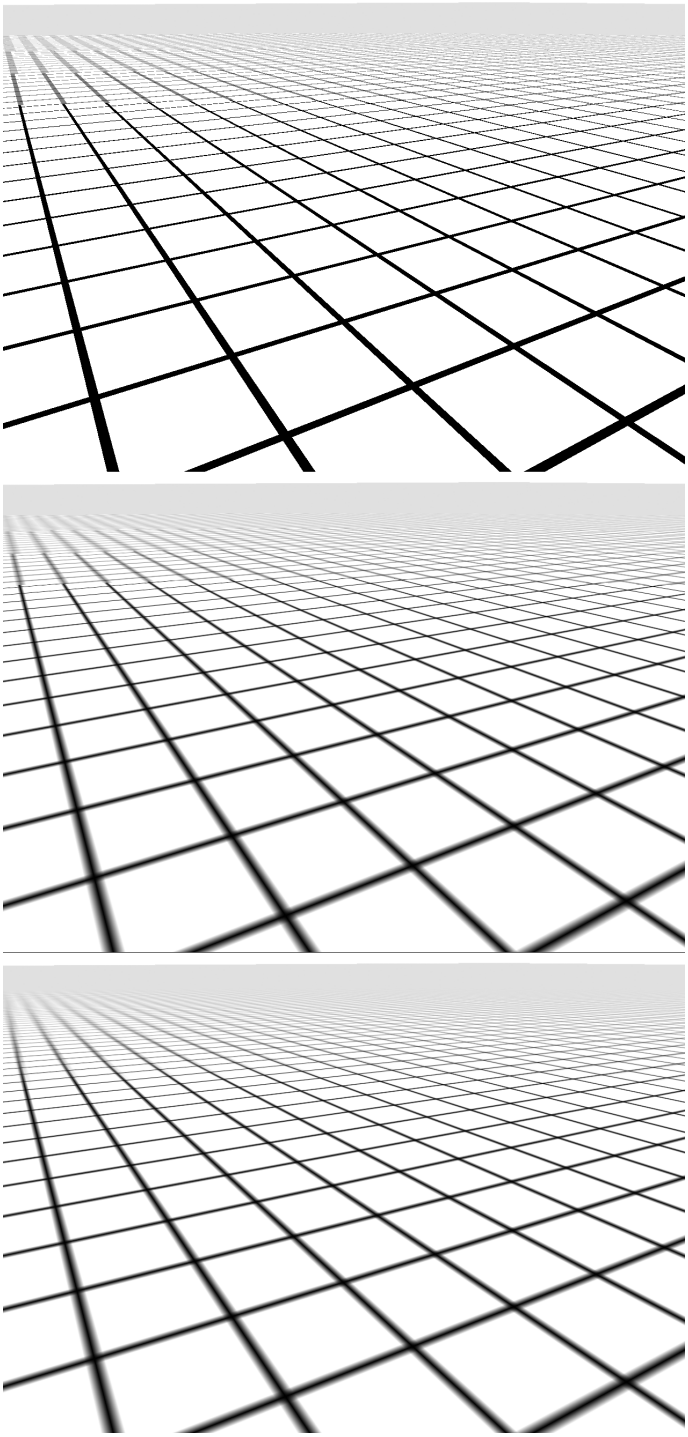


Figura B.9: Comparativa de calidad entre los diferentes filtros implementados: proximidad, bilineal y trilineal.

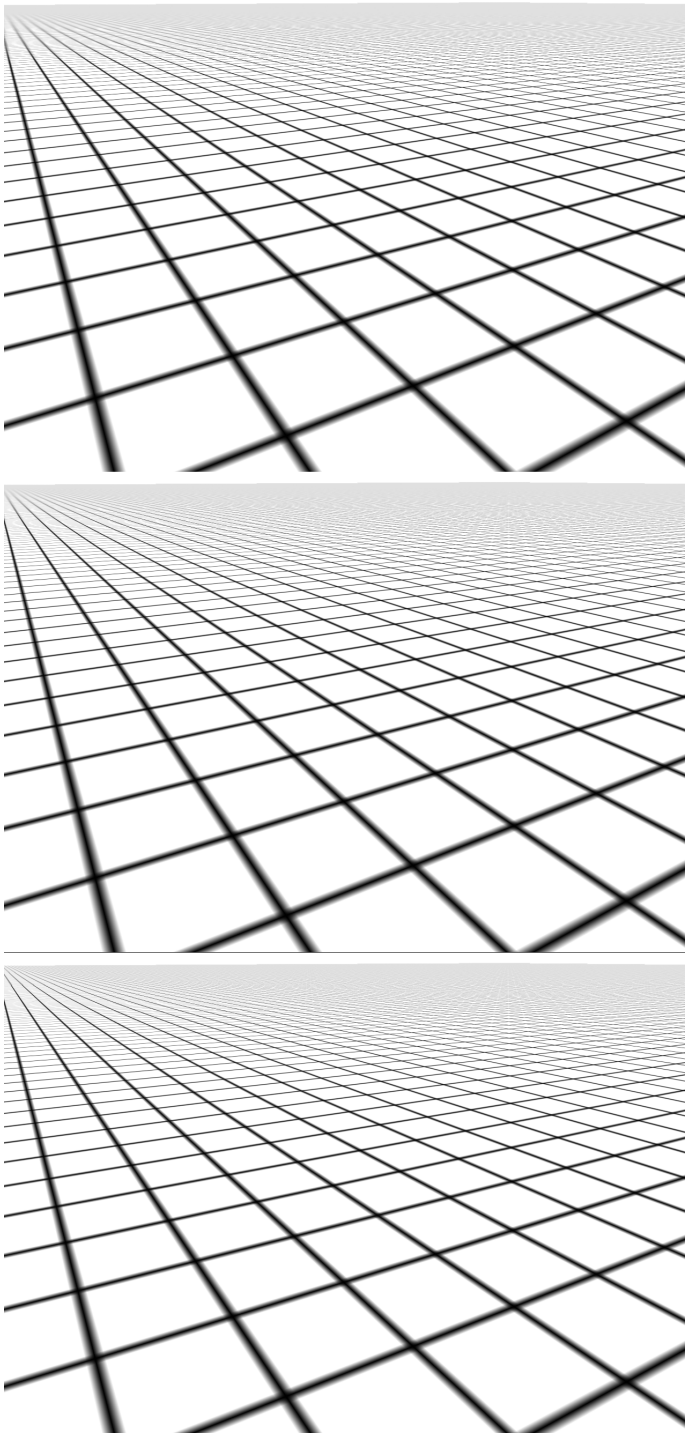


Figura B.10: Comparativa de calidad entre los diferentes filtros implementados: anisotrópico con límites de 4, 8 y 64 muestras.

Apéndice C

Configuración dependiente de la vista

Este apéndice describe una serie de cálculos que se realizan en cada fotograma del *render* para suministrar al motor de texturizado los parámetros necesarios para su correcto funcionamiento. Estos cálculos son dependientes de la vista y se realizan en CPU.

C.1. Ubicación del centro de detalle

Uno de los aspectos críticos para el correcto funcionamiento del sistema, junto con la elección de un tamaño de ventana adecuado y un tamaño de bloque para las transferencias a las cachés (tanto en TRAM como en RAM) que maximice su eficiencia, es la ubicación del centro de detalle.

La ubicación del centro de detalle es una tarea que se debe realizar en el bucle de *render* antes de la fase de actualización de las cachés, descrita en la sección 4.4.

La elección de la posición correcta para maximizar la calidad visual del terreno texturizado no es una tarea trivial, y depende en gran medida del tipo de aplicación.

En concreto, esta elección estará condicionada por la forma en que se moverá la cámara del *render*. A continuación se describen dos situaciones típicas y dos formas de resolver el problema de la colocación del centro de detalle.

C.1.1. Simulación de vuelo

Cuando se trabaja con aplicaciones del tipo de los simuladores de vuelo, el movimiento de la cámara es contínuo y relativamente suave, sin giros excesivamente bruscos y mirando generalmente hacia el horizonte.

En este escenario, la solución habitual es colocar el centro de detalle hacia el interior de la porción de terreno visible frente a la cámara, de forma que se aproveche al máximo el detalle disponible en la caché de textura.

La forma de determinar esta posición es mediante el cálculo de la intersección de uno o varios rayos lanzados desde la cámara hacia el modelo 3D del terreno.

El peligro de esta operación es que si el centro de detalle se aleja demasiado de la posición de la cámara, se puede apreciar una caída de nivel de detalle precisamente en la zona del terreno más cercana, que es la que precisaría la mayor nitidez.

Una forma de enfrentarse a este problema consiste en lanzar rayos coincidiendo con la zona inferior del *frustum* de visión para tomar esta línea sobre el terreno como el límite inferior para la colocación del borde de la ventana cacheada en el nivel de mayor detalle.

En el caso especial de que uno de los ejes del espacio textura esté alineado con el eje horizontal del espacio pantalla, la solución es trivial: se colocaría el borde de la ventana cacheada del máximo nivel alineada con el inferior de la pantalla (figura C.1).

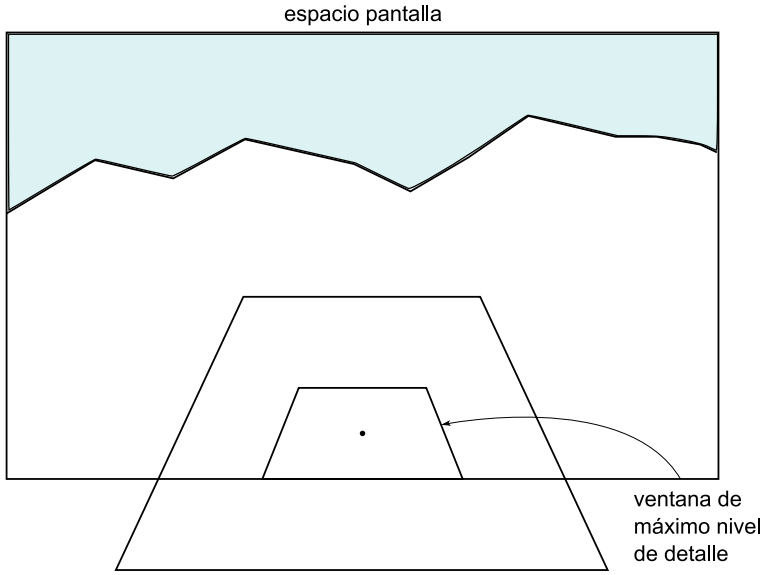


Figura C.1: Ubicación del centro de detalle con la textura alineada con el espacio pantalla.

Sin embargo, la situación habitual es que esto no suceda, y que tengamos un trapezoide proyectado sobre el terreno. En este caso se trata de encontrar un equilibrio entre el aprovechamiento máximo del detalle de textura cacheado frente a la evitación de caídas del nivel de detalle en la zona de terreno más cercana al espectador. La solución estará entre los extremos ilustrados en la figura C.2: colocar la zona completa de máximo detalle dentro del área

visible o atrasar el centro de detalle hacia la cámara hasta maximizar el detalle en la zona más cercana.

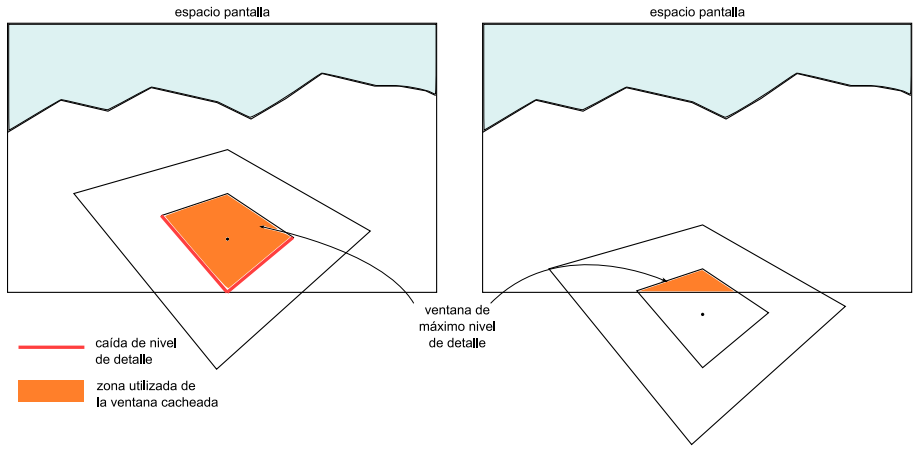


Figura C.2: Ubicación del centro de detalle con la textura no alineada con el espacio pantalla.

La disertación anterior únicamente es cierta si se asume que el terreno es completamente plano, no se ha tenido en cuenta en ningún momento su relieve. Sin embargo, se trata de una aproximación bastante adecuada en la mayoría de los casos. Para evitar las desviaciones en terrenos escarpados y en aquellos casos en que las caídas del nivel de detalle sean apreciables o supongan un problema, se pueden plantear varias soluciones. La más sencilla y barata en tiempo de cálculo es aplicar un pequeño margen de seguridad constante a los cálculos anteriores. Si se necesita mayor precisión y se puede invertir el tiempo necesario para obtenerla, se puede aproximar la desviación (y por tanto el margen de seguridad necesario) mediante el cálculo de varias intersecciones con el terreno en la línea inferior de la pantalla.

Los ejemplos ilustrados anteriormente suponen el peor caso, en el que el tamaño de la ventana cacheada es reducido respecto a la resolución de la pantalla (como se puede apreciar en las figuras C.1 y C.2), situación que se evita en aplicaciones reales.

Con un tamaño de ventana adecuado en la caché de GeoTextura, que tal y como se ha indicado anteriormente debería ser superior a la resolución de la pantalla, el problema desaparece puesto que hay un margen de seguridad más que suficiente para evitar caídas de nivel visibles. Por lo tanto, en estos casos, la segunda opción, que coloca el centro de detalle en la zona inferior de la pantalla, se muestra como la más idónea. Esta solución es la que ha sido utilizada en el visualizador de terreno del proyecto SANTI[83], con un tamaño de ventana de 2048^2 *texels*, sin que en ningún momento se aprecien caídas del nivel de detalle en las zonas del terreno cercanas a la cámara.

Existen algunos casos especiales que se pueden ignorar o tratar por sepa-

rado, lo cual puede ser interesante en caso de uso de un tamaño de ventana reducido por limitaciones en los recursos de *hardware*. El más habitual es cuando en lugar de dirigir la vista hacia el horizonte, se orienta la cámara en un ángulo cercano al cenital. En este caso, no tiene sentido considerar el límite inferior de la pantalla, puesto que no será el más cercano a la cámara. La solución más adecuada en estos casos es colocar el centro de detalle en la posición de la intersección con el terreno de un rayo trazado a través del centro de la pantalla, o para un resultado más preciso, el punto medio de la posición de las intersecciones de los rayos trazados a través de las cuatro esquinas de la pantalla.

En caso de que la vertical de la cámara se desvíe excesivamente de la del terreno (por ejemplo, si la cámara se da la vuelta, volando "cabeza abajo"), tampoco serviría el cálculo descrito, basado en que la zona inferior de la pantalla es la más cercana. En este caso la mejor opción sería adaptar el algoritmo para que considere la cercanía de las cuatro esquinas de la pantalla y coloque la ventana de la caché de textura próxima a la zona más cercana, de forma similar a la descrita anteriormente.

C.1.2. Sistemas de realidad virtual

Los simuladores de vuelo no son la única aplicación del motor de textura descrito. Otra situación posible, relativamente habitual, sobre todo en el ámbito de la investigación, es el uso de sistemas de realidad virtual para observar el terreno.

Uno de los dispositivos más característicos son los denominados HMD, del ingles *head mounted display* o pantallas montadas en la cabeza del usuario (y situadas ante sus ojos, por supuesto).

La otra familia de dispositivos utilizados en aplicaciones de realidad virtual son los de proyección inmersiva, tipo CAVE [59, 58].

En ambos casos, el usuario tiene total libertad para observar hacia cualquier posición dentro del espacio 3D, de forma que se ve inmerso en ese escenario virtual.

El problema de esta situación, en contraposición a un simulador de vuelo, es que los usuarios pueden cambiar la orientación de la visión de forma casi instantánea, con algo tan natural como un rápido giro de cabeza. Desplazar el centro de detalle y actualizar la caché de la textura en esta situación supone unas necesidades de velocidad de transferencia muy elevadas. Por este motivo, la solución más habitual es colocar el centro de detalle directamente bajo la cámara. El cálculo no puede ser más sencillo: se toma directamente la posición 2D de la cámara en el plano horizontal, descartando la altura. Se dispone del mismo nivel de detalle en todas direcciones, por lo que los giros del usuario no supondrán ni pérdidas de calidad ni cambios visibles en la nitidez de cada zona.

C.2. Ubicación de la pirámide flotante

Cuando se trabaja con texturas de gran tamaño, de forma que se supere el límite de direccionamiento de un número representado en formato de coma flotante de precisión simple (32 bits), es necesario, al acercarse al terreno para apreciar un detalle más fino, proporcionar las coordenadas de textura al sistema gráfico tomando como espacio textura un subconjunto de la extensión total de la textura virtual y definir también el número de niveles que se manejarán (posición de la pila flotante o ventana de niveles cacheada dentro de la pila virtual). Esta situación ha sido descrita en la sección 4.7.

La selección del espacio de terreno direccionado dentro de la textura virtual se puede realizar automáticamente por parte del motor de textura a partir de la posición del centro de detalle. De esta forma se garantiza que la ventana cacheada esté contenida dentro la zona direccionable.

La tarea de selección de la pila flotante debería por tanto realizarse junto con la selección del centro de detalle. Ambas están muy relacionadas entre ellas y es necesario realizarlas antes de la actualización de la caché, puesto que determinarán completamente su comportamiento.

En la sección anterior se describió la problemática de la colocación del centro de detalle, y junto con él la ventana de la pirámide virtual que se considera por límites en la capacidad de direccionamiento. El parámetro que queda por determinar es el intervalo de niveles de textura que van a estar disponibles en la caché, es decir, la ventana de niveles que se cachean de la textura virtual, que junto a los límites de direccionamiento define una subpirámide virtual (figura 4.30), y que se ha denominado pila flotante.

La pila flotante se ubica dentro de los niveles de la textura virtual mediante el parámetro desplazamiento respecto al nivel inferior de la pila virtual. Este desplazamiento de la pila flotante se debe establecer antes de la fase de actualización, junto con la posición del centro de detalle.

El objetivo que se debe perseguir en el cálculo de la ubicación de la pila flotante es que el máximo nivel cacheado corresponda con el máximo nivel necesario en el *render*. Adicionalmente, en caso de aproximación al detalle, se puede añadir un margen de seguridad, de forma que se precarguen algunos niveles de mayor detalle y estén así disponibles en caché en el momento en que sean necesarios en el *render*.

Es importante ser cuidadoso con este margen de seguridad, puesto que si se eleva la pila flotante más allá de lo necesario, se pueden llegar a descargar niveles de textura inferiores que hagan falta para las zonas lejanas de la escena. Manteniendo próxima la cima de la pila flotante al máximo nivel necesario en el *render*, los niveles disponibles en dicha pila flotante deberían ser suficientes para un correcto texturizado del terreno visible.

El algoritmo utilizado para determinar la posición de la pila flotante en el prototipo implementado en este trabajo se describe a continuación.

En primer lugar, se calcula la distancia entre la posición de la cámara y

el punto del terreno más cercano a la pantalla. Se pueden utilizar diferentes aproximaciones según la disponibilidad de recursos, tal y como se comentó en el caso del cálculo del centro de detalle. La opción más simple puede ser lanzar un rayo de intersección con el terreno a través del centro de la pantalla o del borde inferior, o utilizar mejores aproximaciones lanzando varios rayos, calculando sus intersecciones y tomando la muestra más cercana.

Una vez obtenida esa distancia (d), se calcula la razón entre la resolución de la pantalla en pixels en un eje y el espacio visible a esa distancia (en unidades del espacio mundo, habitualmente metros) para ese mismo eje. Si se asume que los pixels son cuadrados, cuál de los dos ejes se tome para los cálculos es irrelevante.

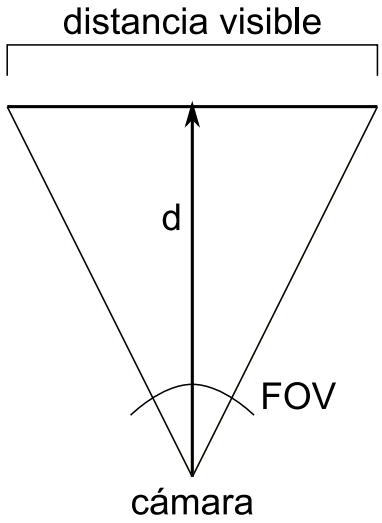


Figura C.3: Cálculo de la distancia visible en pantalla (en unidades del espacio mundo).

La distancia visible en espacio mundo se calcula a partir del ángulo de visión o FOV (del inglés *field of view*), y la distancia d anteriormente calculada al punto del terreno visible más cercano a la cámara (figura C.3), mediante la ecuación C.1.

$$v = 2d \left| \tan \frac{FOV}{2} \right| \tag{C.1}$$

Finalmente, conociendo la resolución (res) en el mismo eje de la cámara cuyo ángulo de visión se tomó, ya se puede calcular la cota inferior de la ratio de pixels por metro (pm) de la zona visible en pantalla (ecuación C.2).

$$pm = \frac{res}{v} \tag{C.2}$$

Esta ratio se compara con la de *texels* por metro (tm) del máximo nivel de la textura para ver cuántos niveles por debajo de este máximo estará la

cima de la pila flotante que se necesita para la posición actual de la cámara (ecuación C.3). Este desplazamiento de la cima de la pila flotante (*dcp*) se comunicará al objeto CacheTRAM para que actúe en consecuencia.

$$dcp = \left| \log_2 \frac{mt}{mp} \right| \tag{C.3}$$

Se obtiene de esta forma una cota mínima para el desplazamiento de la cima de la pila flotante, calculada con unos criterios conservadores, de forma que se garantiza que siempre estará disponible el nivel necesario (y según los casos, puede que algún nivel a mayores).

La transformación del desplazamiento desde la cima de la pila (d_c) al desplazamiento desde la base de la pila (d_b) es trivial, y se realiza mediante la ecuación C.4, donde n es el máximo nivel de la pila virtual y n_d el máximo nivel direccionable (máximo nivel de la pirámide flotante).

$$d_b = \text{máx}(\text{mín}(n - n_d - d_c, n - n_d), 0) \tag{C.4}$$

C.3. Cálculo del nivel de detalle principal

El nivel de detalle principal se utiliza para determinar el orden de carga de los niveles en las texturas dinámicas con ciclo de actualización breve, tal y como se describe en la sección 4.4.1.

Este nivel principal es el mismo que se calcula para establecer la cima de la pirámide flotante, de forma que su cálculo utiliza las ecuaciones descritas en el apartado anterior. Simplemente restando el valor *dcp* obtenido de la ecuación C.3 del máximo nivel de la pirámide virtual se obtiene el nivel principal para el algoritmo de ordenación de carga de niveles.

Apéndice D

Escalabilidad del sistema

En este apéndice se detallan las modificaciones que es necesario realizar sobre el sistema descrito hasta el momento, para tener en cuenta y solucionar los problemas de escalabilidad planteados en la sección 4.7. Estas modificaciones se han clasificado en tres apartados: arquitectura e interfaz del motor de texturizado, proceso de *render* y proceso de actualización.

D.1. Modificaciones a la arquitectura y la interfaz

Desde el punto de vista del cliente, se añade una nueva tarea que se debe realizar junto con la selección del centro de detalle. Se trata de la selección de la ubicación de la pila flotante dentro del espacio de la pila virtual.

Para esta tarea, se ha desarrollado un módulo que realiza dicho cálculo automáticamente a partir de los siguientes parámetros, tal y como se describe en el apendice C.2.

- Distancia de la cámara al punto más cercano del terreno o en su defecto una aproximación.
- Ángulo de visión de la cámara (en uno de los dos ejes: horizontal o vertical).
- Resolución en pixels de la ventana de *render* en el mismo eje que el ángulo de visión.

La gestión de la pirámide flotante supone una sobrecarga adicional a los cálculos que realiza el sistema, por lo que sólo se activará cuando sea necesario. A continuación se describen las modificaciones necesarias en diferentes componentes de GeoTextura.

El objeto CacheTRAM detectará durante la asignación de la base de datos (método `CacheTRAM::setBD`) si el tamaño de la pila necesaria para dicha base de datos supera el tamaño máximo de pila impuesto, bien por

el usuario, bien por limitaciones de memoria o bien por limitaciones en el direccionamiento. En caso de superarse dicho tamaño, se activa el mecanismo de gestión de pila virtual, se inicializan los parámetros necesarios para dicha gestión y se notifica al CargadorTRAM para que direccione adecuadamente la pila en la textura 3D.

Se añaden nuevos métodos a la clase CacheTRAM para obtener información acerca del estado de la pila flotante y el área direccionable (pirámide flotante) así como para establecer el desplazamiento de dicha pila.

El área direccionable, que corresponde con la pirámide flotante, se calculará en función del desplazamiento de la pila flotante y de la posición del centro de detalle. CacheTRAM es responsable de realizar dicho cálculo cada vez que se modifique el centro de detalle o el desplazamiento de la pila flotante.

A continuación se describirán las modificaciones específicas que afectan a los procesos de *render* y actualización previamente descritos, debidas a la gestión de la pirámide flotante.

D.2. Modificaciones al proceso de *render*

El proceso de *render* descrito anteriormente sufre ciertas modificaciones debido al uso de la pila virtual y la ventana de direccionamiento de la pirámide flotante.

El primer aspecto a tener en cuenta es que las coordenadas de textura se deben suministrar al sistema gráfico ya no en el espacio completo de la textura virtual, sino en el área definida como direccionable. Este área se determinará en función del nivel de mayor detalle de la pila flotante y de la posición del centro de detalle.

El área direccionable se suministra al *vertex shader* a través del parametro de tipo *uniform* `areaTex`, tal y como se ha descrito, de manera que los cálculos se pueden realizar de forma equivalente.

Los parámetros suministrados al *fragment shader*, en concreto los utilizados por la función `calculaFragment` y sus funciones auxiliares, que sufren alguna modificación o que se añaden por ser necesarios para la gestión de la pirámide virtual se describen a continuación.

MaxNivelPila (macro) El nivel máximo de la pila indicado al *shader* corresponderá al nivel máximo de la pila o la pirámide flotantes, que será el espacio que se considerará para todas las operaciones de direccionamiento.

TamPila (macro) El tamaño de la pila indicado será el tamaño físico de la pila. Este valor es necesario para acceder a la textura 3D siguiendo el direccionamiento circular descrito (figura 4.31).

limites (*uniform*) Esta tabla de límites de información válida en cada nivel tendrá el tamaño de la pila física `TamPila`, y las coordenadas de los límites corresponderán al espacio direccionable de la pirámide flotante, no al espacio de la textura completa.

tamTex (*uniform*) El tamaño de la textura en *texels* se ajusta al tamaño cubierto por la pirámide flotante.

maxLOD (*uniform*) El valor de máximo nivel con información útil se ajusta al máximo nivel direccionable en caso de que fuese superior a éste.

desplazamientoLocalPila (*uniform*) Este nuevo parámetro suministra el desplazamiento local necesario para el direccionamiento circular de la pila (figura 4.31).

desplazamientoGlobalPila (*uniform*) Este nuevo parámetro suministra el desplazamiento global de la pila flotante (figura 4.31).

desplazamientoAreaDireccionableNormalizado (*uniform*) Este nuevo parámetro suministra el desplazamiento del área direccionable de la pirámide flotante, indicado en coordenadas normalizadas del espacio de dicha textura. Se utilizará este desplazamiento para la correcta ubicación de la textura obtenida de las capas de la textura 3D.

Las operaciones realizadas por el *shader* resultan adecuadas para la gestión de la pirámide flotante con las modificaciones anteriores en los parámetros y con las modificaciones que se indican a continuación en el código GLSL.

En la función `calculaNivel`, tras la comprobación de disponibilidad de la textura en los niveles solicitados, y antes del resto de transformaciones a las coordenadas de textura, se les debe aplicar a éstas el desplazamiento debido a la posición del área direccionable de la pirámide flotante dentro del espacio completo de la textura.

En el caso de los niveles de la pila, este cálculo resulta trivial: simplemente se suma el desplazamiento del área direccionable normalizado respecto a la propia área direccionable.

En los niveles de la pirámide, en cambio, hay que tener en cuenta que la pirámide disponible no es la inmediatamente inferior a la pila flotante, sino que se trata de la pirámide inferior de la textura virtual completa. Esto implica que hay un salto de niveles entre la pila y la pirámide disponibles, correspondiente al parámetro `desplazamientoGlobalPila`. Este salto de niveles requiere un escalado de las coordenadas de textura tras aplicar el desplazamiento del área direccionable según se detalla en el código siguiente.

```
tex_coord += tex0_desplazamientoAreaDireccionableNormalizado;
if ( nivel <= MaxNivelPiramide )
    tex_coord /= pow(2.0, tex0_desplazamientoGlobalPila);
```

En la misma función, antes de direccionar la textura 3D, y en caso de tratarse de un nivel de la pila, se debe ajustar la capa de dicha textura a la que se accede teniendo en cuenta el direccionamiento circular de la pila flotante (ver figura 4.31).

```
float slice;
if ( nivel > MaxNivelPiramide ) {
    slice = mod( nivel - MinNivelPila +
        desplazamientoLocalPila, TamPila ) + 2.5;
}
else {
    slice = max( nivel - MaxNivelPiramide + 1.5, 0.5 );
}
```

Como se ha indicado, la pirámide disponible realmente no corresponde con la que contendría la pirámide flotante, situada inmediatamente a continuación de la pila flotante seleccionada. Actualizar esta pirámide supondría unas transferencias de textura bastante ineficientes, puesto que se trata de muchos niveles de tamaño reducido, para una información muy poco importante. La información de textura correspondiente a la pirámide sólo se utilizará en zonas muy lejanas, en caso de que llegue a usarse en absoluto. Si la selección de la pila flotante es acertada, su uso será en muy contadas ocasiones, correspondiendo a determinadas circunstancias: vista al horizonte, terreno muy plano y con un límite de visibilidad muy elevado. En la práctica se ha llegado a apreciar con un terreno plano (elevación 0 en toda la superficie) y sin límites a la distancia visible[1] únicamente con filtros isotrópicos.

El filtrado anisotrópico aumenta el nivel de detalle de las muestras que se toman, por lo que se reducirán o eliminarán los accesos a los niveles inferiores, correspondientes a la pirámide. En las pruebas realizadas durante este trabajo no se apreció falta de detalle en el horizonte en ningún caso aún en las condiciones más extremas.

Sin embargo, para gestionar correctamente aquellos casos en que se acceda a la pirámide, hay que tener en cuenta un último detalle a mayores de su correcto direccionamiento que ya fue expuesto anteriormente. Se trata del cálculo del nivel de detalle a aplicar, que en los niveles de la pila será correcto, pero en los niveles de la pirámide el nivel calculado corresponde a la pirámide contigua a esa pila, que tiene un cierto desplazamiento que la sitúa algunos niveles más arriba en la pirámide real.

En la figura D.1 se ilustra esta situación. En el caso de desplazamiento cero el comportamiento es el habitual (primera gráfica). Si el desplazamiento es inferior al número de niveles de la pirámide se produce la situación ilustrada en la segunda gráfica, donde se aprovecha un subconjunto de los niveles

[1] Es habitual que se limite la zona visible por motivos de rendimiento, y por que así sucedería en la realidad, disimulando el corte mediante el uso de niebla simulada.

superiores de la pirámide. Si el desplazamiento excede el número de niveles de la pirámide, únicamente se usa el nivel superior, tal y como se ilustra en la tercera gráfica[2].

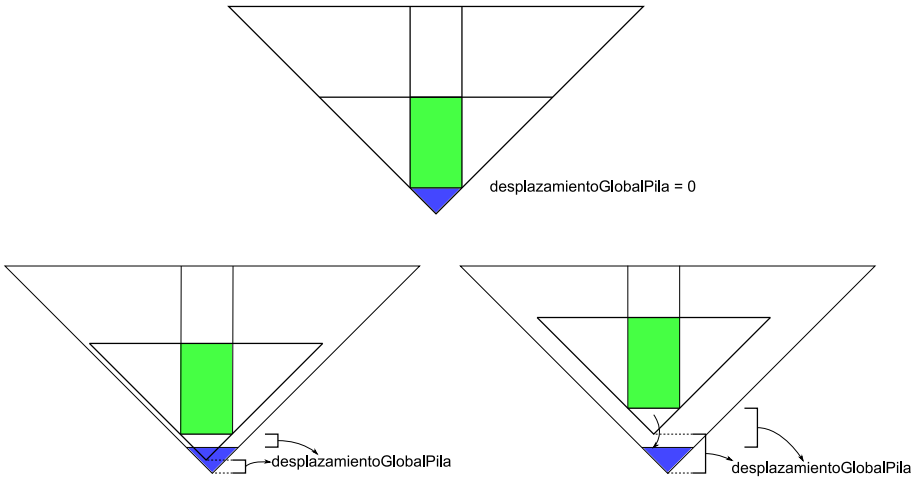


Figura D.1: Desplazamiento al nivel de la pirámide direccionado.

Por lo tanto, cuando el mecanismo de pila flotante está activado, en caso de solicitar niveles de la pirámide, se incrementa el número de nivel en el desplazamiento de la pila flotante, sin superar en ningún caso el nivel máximo disponible en la pirámide. Este desplazamiento se aplicará después de todos los ajustes realizados al nivel de detalle, descritos anteriormente.

```
if ( nivel <= MaxNivelPiramide )
    nivel = min( nivel + tex0_desplazamientoGlobalPila,
        MaxNivelPiramide );
```

D.3. Modificaciones a la actualización

El proceso de actualización no se ve gravemente afectado por el sistema de pirámide virtual. Las estructuras que almacenan los estados de actualización de las teselas y niveles de la caché permanecen exactamente iguales, trabajando en el espacio de la pirámide virtual completa.

Al realizar las tareas periódicas de actualización, se tiene en cuenta el salto de niveles entre la pirámide fija en los niveles inferiores absolutos y la pila flotante, que se puede separar de esta un número determinado de

[2]Esta última situación, sin embargo, es bastante infrecuente: la mayor base de datos utilizada para las pruebas corresponde a la textura de extensión planetaria con un detalle máximo de 0,25 metros/texel que resulta en una pirámide completa de 27 niveles; limitando la pila a 22 niveles, únicamente se obtendrá un desplazamiento máximo de 5 niveles, mientras que la pirámide, con un tamaño de ventana cacheada de 2048×2048 *texels*, dispone de 11 niveles.

niveles. Esto implica que la actualización se hace en dos pasadas, tratando el intervalo de niveles de la pirámide y el de la pila por separado aunque por lo demás las tareas son equivalentes.

Otro aspecto de vital importancia es que la información proporcionada al *render* se debe transformar del espacio global al espacio direccionable de la piramide flotante. En concreto se trata de los límites de las áreas rectangulares de cada nivel con información válida, suministradas al *shader* como el parámetro de tipo *uniform* `limites`.

Esta transformación se realiza mediante las operaciones indicadas en la ecuación D.1, donde v es cualquiera de los vértices del área válida en espacio virtual, v' es el mismo punto transformado al área direccionable, d es el desplazamiento del área direccionable dentro del área virtual, t es el tamaño del área virtual y p es el desplazamiento de la pila flotante. En la figura D.2 se ilustra gráficamente el significado de los términos de la ecuación.

$$v' = \left(v - \frac{d}{t}\right) 2^p \tag{D.1}$$

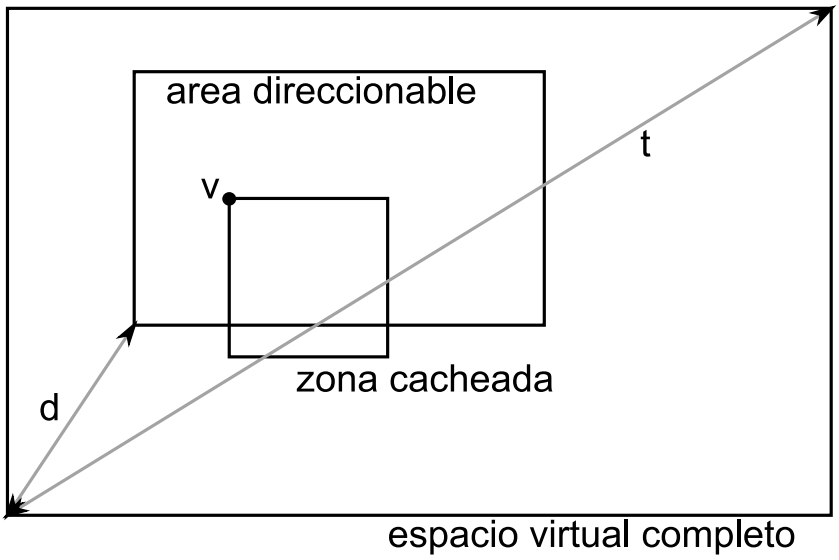


Figura D.2: Transformación de las coordenadas de los límites de la zona válida cacheada desde el espacio virtual completo al espacio direccionable por el *shader*.

Durante la carga de teselas a la caché, el objeto CargadorTRAM debe colocar las correspondientes a la pila flotante en el lugar correcto de la TRAM. Por este motivo, es informado por el objeto CacheTRAM que lo controla de la activación del mecanismo de pila flotante y del desplazamiento actual.

Los objetos de la clase CargadorTRAM deben conocer la situación de la pila flotante para realizar las cargas en el lugar correcto de la caché (en el

caso de la implementación CacheTRAM3D, en la capa correcta de la textura 3D). CacheTRAM les informará por lo tanto de la activación del mecanismo de gestión de pila virtual, cuando proceda, junto con el desplazamiento de la pila flotante y el tamaño de dicha pila flotante almacenada físicamente en TRAM a través de nuevos métodos añadidos a la clase CargadorTRAM.

Apéndice E

CargadorTRAM

`CargadorTRAM` es la pieza encargada de suministrar las teselas que se almacenarán en TRAM, ya sea mediante su carga desde RAM o bien mediante su generación en la GPU a partir de datos vectoriales (*render* a textura).

Durante este trabajo se han desarrollado varias implementaciones de CargadorTRAM, de las cuales se utilizan finalmente dos: una para la carga de datos *raster* desde memoria principal (`CargadorTRAM3DTexSubImage`) y otra para el *render* a textura de datos vectoriales mediante OpenGL (`CargadorTRAM3DRender`). Ambos cargadores están diseñados para funcionar con la implementación de `CacheTRAM` basada en texturas 3D (`CacheTRAM3D`), aunque serían fácilmente adaptables a otras implementaciones.

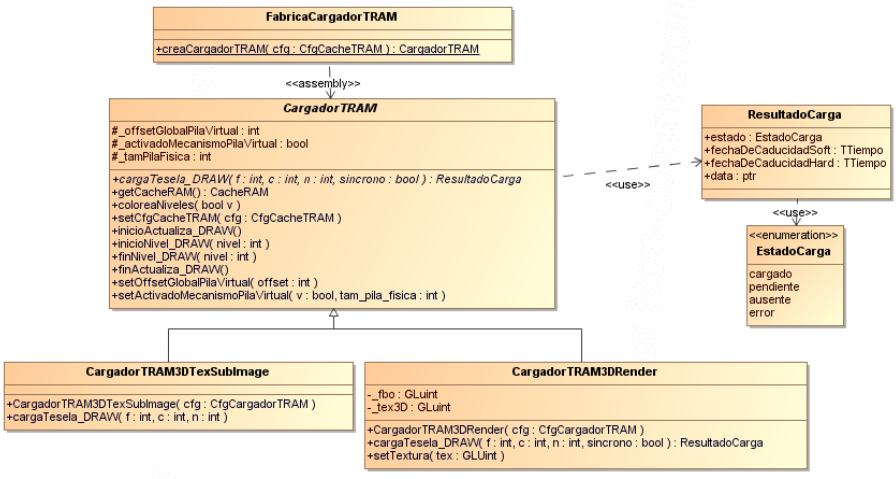


Figura E.1: Diagrama UML de la clase `CargadorTRAM`.

La interfaz común a ambos cargadores, ilustrada en la figura E.1, es muy simple y se basa en la división del espacio virtual de cada nivel de detalle en teselas. Para solicitar una tesela se indican sus coordenadas en el espacio virtual (fila, columna y nivel) así como si se desea que la carga se realize de

forma síncrona o asíncrona. En caso de no estar disponibles los datos solicitados en el segundo nivel de caché (`CacheRAM`), la carga asíncrona devolverá el control inmediatamente, mientras que una carga síncrona esperará a que esos datos estén disponibles en la caché o se indique una condición de error o ausencia de los mismos antes de devolver el control.

Habitualmente se realizarán cargas asíncronas para no afectar al rendimiento interactivo del sistema, pero en algunos casos excepcionales es necesario disponer del mecanismo para solicitar cargas síncronas. Un ejemplo de la utilidad de estas cargas síncronas puede ser una fase de precarga de la caché durante el arranque del sistema de forma que se comience con la máxima calidad disponible sin tener que esperar un cierto tiempo a que se cargue progresivamente la textura hasta alcanzar el máximo detalle.

E.1. Datos *raster*. CargadorTRAM3DTexSubImage

Para la carga de información *raster* a textura desde memoria principal, existen varias aproximaciones. Una de las más utilizadas, disponible desde las primeras versiones de OpenGL es la función `glTexSubImage`, que permite sustituir la información de una región de una textura ya existente. Esto es exactamente lo que se necesita hacer en la textura utilizada para la caché en TRAM.

Es importante la diferencia de rendimiento entre las llamadas glTexImage y glTexSubImage aún cuando se utilicen para cargar la información de la textura completa, por lo que es muy importante el uso de esta última sobre una textura ya creada.

Posteriormente se hicieron disponibles otras alternativas como el uso de PBOs (del inglés *pixel buffer objects*). Sin embargo, debido a la arquitectura diseñada que mantiene un conjunto de buffers en RAM para la caché de segundo nivel (CacheRAM) de los cuales se solicitan los datos para cargar las teselas, el uso de PBOs no ofrece mejora alguna. Los PBOs resultarían de utilidad en caso de unificar las cachés de TRAM y RAM, de forma que los buffers/teselas se actualizasen direccionándolos directamente desde la CPU. De hecho en una implementación del cargador utilizando PBOs se ha comprobado que efectivamente incluso se pierde algo de rendimiento en comparación al cargador basado en la llamada `glTexSubImage`.

La carga de información *raster* desde memoria principal a memoria de texturas se realiza mediante objetos de la clase `CargadorTRAM3DTexSubImage`. El funcionamiento de esta clase es bastante directo, y comprende las siguientes fases.

- Calcular la zona de la textura 3D correspondiente a la tesela a cargar. Se debe tener en cuenta, entre otras cosas, la continuidad de la textura en ambos ejes y el direccionamiento de pilas o pirámides flotantes, descritas en la sección 4.7.

- Solicitar la información correspondiente a esa zona a la caché de segundo nivel (`CacheRAM`)
- En caso de obtener la información de la caché de segundo nivel, se transfiere la información necesaria a memoria de textura mediante la función `glTexSubImage` estableciendo previamente los estados OpenGL adecuados.
- Finalmente se devuelve el estado de la carga, que puede ser uno de los descritos en la sección de actualización: cargado, pendiente, ausente o error.

Una de las limitaciones más importantes descubiertas durante este desarrollo es que con las texturas descomprimidas en GPU, mediante la técnica de compresión S3TC (DXT1 a DXT5), no es posible actualizar una subregión de una textura 3D, sólo se puede transferir la textura completa. Con texturas 2D es posible transferir subregiones siempre y cuando no haya discontinuidades en la zona transferida. En otras palabras, con las texturas en formato S3TC se ignoran los modos de transferencia de pixel `GL_UNPACK_SKIP_ROWS` y `GL_UNPACK_SKIP_PIXELS` y `GL_UNPACK_ROW_LENGTH` debe corresponder con el ancho de la textura.

Esta limitación se puede solventar con el uso de *arrays* de texturas[121] en lugar de texturas 3D, y así se realizó en la implementación de este trabajo.

E.2. Datos vectoriales. CargadorTRAM3DRender

La representación de datos vectoriales mediante textura implica inevitablemente un proceso de *rasterización*. Esta *rasterización* se puede realizar a diferentes niveles.

Una de las posibilidades es hacerlo en el origen de datos, de forma que los datos ya se cargan en la caché en RAM en formato *raster*. Este comportamiento se utiliza por ejemplo cuando se accede mediante el sistema descrito en este trabajo a fuentes de datos vectoriales a través del servicio WMS.

La otra alternativa, más potente, versátil y eficiente, consiste en realizar el *render* bajo demanda, en tiempo real, en la GPU. Es decir, el `CargadorTRAM` en lugar de realizar una transferencia de la información, genera la propia información sobre la textura a partir de los datos vectoriales.

Para esta tarea se utiliza la clase `CargadorTRAM3DRender`, cuya estructura y comportamiento se describen a continuación.

El funcionamiento de este cargador se basa en el *render* a textura mediante OpenGL, por lo que utiliza el mecanismo *frame buffer object* (FBO) estableciendo la textura de la caché como destino del *render*. `CargadorTRAM3DRender` necesita por lo tanto conocer la textura 3D de OpenGL que contiene la caché, por lo que se añade a la interfaz de `CargadorTRAM` un método `setTextura` para que `CacheTRAM3D` le pueda indicar cuál dicha textura.

`CargadorTRAM3DRender` trabaja en conjunto con una caché de segundo nivel específica para el manejo de información vectorial, definida en la subclase `CacheRAMSceneGraph` de `CacheRAM`.

En la inicialización (`init_DRAW`), dentro del contexto gráfico, se construye el FBO que se utilizará para hacer el *render* a textura. Además, se comprueba también si el *hardware* soporta las operaciones necesarias para hacer el *render* sobre la textura 3D. En el *hardware* utilizado durante el desarrollo se comprobó que estas operaciones no funcionan en tarjetas Nvidia GeForce anteriores a la serie 8.

Para minimizar el número de operaciones y cambios de estado en la máquina de OpenGL, con el objetivo de obtener un rendimiento óptimo, se hace uso del mecanismo de notificación de comienzo y fin de la actualización descrito en la sección 4.4, tanto a nivel global como para cada nivel de detalle. De esta forma se garantiza que se preservan después de la actualización los estados de OpenGL previos a la misma, con un coste mínimo.

Al comienzo de la actualización (`inicioActualiza_DRAW`) se realizan las siguientes operaciones:

- Almacenar en la pila de matrices de OpenGL el estado de las matrices `GL_MODELVIEW` y `GL_PROJECTION`.
- Almacenar los estados relativos a la ventana (*viewport*) de *render* (`GL_VIEWPORT_BIT`).
- Activar el FBO para que las subsecuentes órdenes de dibujado lo tengan como destino en lugar de la ventana de *render* de la aplicación.

Al comienzo de la actualización para un nivel de detalle determinado (`inicioNivel_DRAW`) se calcula la capa (*slice*) de la textura 3D que contiene a dicho nivel y se asocia esa capa de la textura al FBO para tomarla como destino del *render* para los canales de color.

Al final de la actualización (`finActualiza_DRAW`) se desactiva el FBO para volver a la ventana de *render* de la aplicación y se restauran los estados de las matrices de OpenGL `GL_MODELVIEW` y `GL_PROJECTION` y de los atributos relativos a la ventana de *render* (`GL_VIEWPORT_BIT`).

El proceso de carga de una tesela por parte de la clase `CargadorTRAM3DRender` delega en la caché en RAM, que es quien contiene la información vectorial, la responsabilidad de la tarea de *render*. Únicamente se establecen los estados de OpenGL necesarios para que el *render* se produzca en la zona correcta de la textura de la caché en TRAM. La carga se compone de las siguientes operaciones:

- Calcular la ventana correspondiente a la tesela y establecerla como ventana de *render* (`gl_Viewport`).

- Borrar el *frame buffer* mediante la orden `gl_Clear`. Se inicializan tanto los canales de color como el canal alfa utilizado habitualmente para definir la transparencia u opacidad. No se borra el canal de profundidad (Z o *depth buffer*), puesto que para el *render* de la información vectorial en 2D no se utilizará. Para asegurarse de que únicamente se borra la tesela que se va a procesar, sin afectar a otras teselas de esa capa de la textura 3D de la caché, se establece previamente la misma región que se estableció en el *viewport* como límites de recorte de OpenGL (`gl_Scissor`) y activar dicho recorte (`GL_SCISSOR_TEST`).

- Solicitar a `CacheRAMSceneGraph` que envíe a OpenGL las órdenes de *render* correspondientes a la información vectorial contenida en la tesela.

En resumen, `CargadorTRAM3DRender` se encargará de guardar y restaurar los estados de OpenGL afectados, gestionar el FBO, establecer la capa correcta de la textura 3D como destino de *render* para cada nivel de detalle, configurar la ventana de *render* para la tesela y solicitar a `CacheRAMSceneGraph` el *render* de la tesela, delegando en éste las operaciones de establecer en OpenGL las matrices de vista, modelo y proyección y enviar las órdenes de dibujado de la información vectorial.

Los detalles de la gestión de datos vectoriales para su correcta visualización sobre la textura se describen en la sección 4.6.

Apéndice F

CacheRAM

La misión del componente `CacheRAM` es almacenar en memoria la información, ya sea *raster* o vectorial, que se representará en la textura virtual y proporcionar al componente `CargadorTRAM` el acceso a dicha información.

El comportamiento será bastante diferente según se trate de datos *raster* o vectoriales. En este último caso se utiliza una subclase de `CacheRAM` denominada `CacheRAMSceneGraph`.

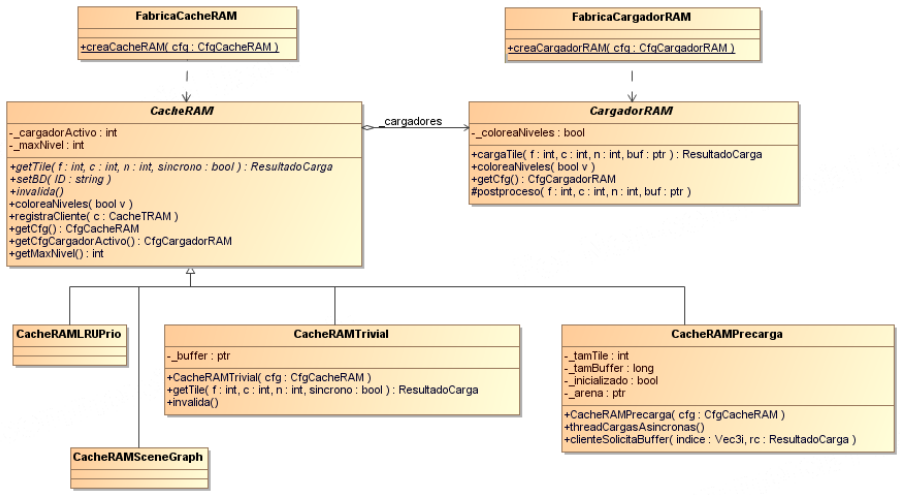


Figura F.1: Diagrama UML de la clase `CacheRAM`.

F.1. Cachés de información *raster*

Para el caso de datos *raster*, la caché se basa en una colección de *buffers* reservados en memoria principal para albergar una zona de la textura adecuada para alimentar la caché en TRAM. Se han desarrollado varias implementaciones de `CacheRAM` destinadas al manejo de información *raster*, algunos ejem-

plos de éstas son `CacheRAMTrivial`, `CacheRAMLRUPrio` y `CacheRAMPrecarga`.

La primera es una implementación sencilla y absolutamente ineficiente, que almacena únicamente un *buffer* y realiza una carga síncrona cada vez que se solicita. Su utilidad es facilitar la depuración y las pruebas de unidad de los otros componentes del sistema, al eliminar la complejidad de las otras implementaciones de caché en RAM que crean hilos de ejecución y añaden la necesidad de sincronización y gestión de la concurrencia en el acceso a los datos.

La segunda implementación, `CacheRAMLRUPrio`, es una caché que asigna prioridades a los diferentes niveles de menor a mayor detalle y utiliza un sencillo algoritmo LRU para la gestión de los buffers asignados a nuevas cargas. Esta implementación, utilizada en algunas aplicaciones, fue sustituida por la última (`CacheRAMPrecarga`), que realiza una carga predictiva en función de la configuración de la caché de la textura virtual y el desplazamiento del centro de detalle. El comportamiento de este sistema de caché predictiva está descrito en la sección 4.4.

La caché en RAM puede ser compartida por varios clientes (`CacheTRAM`/ `CargadorTRAM`) para facilitar su reutilización en aplicaciones con varias vistas de la misma escena. En estos casos, si cada vista tiene una ubicación diferente del centro de detalle, debe tener una caché en TRAM propia, pero por rendimiento y economía de recursos, la caché en RAM debe ser compartida.

Por este motivo, `CacheRAMPrecarga` almacena varias colecciones de *buffers* correspondientes a las zonas A y B de diferentes clientes, las cuales pueden estar solapadas. Para mantener esos *buffers* críticos (zonas A y B) cargados en memoria y evitar que se reutilicen para cargar nueva información cuando dejen de ser necesarios en alguno de los clientes, se utiliza un mecanismo de contador de referencias asociado a cada *buffer*, de forma que un *buffer* no se libera hasta que haya salido de las zonas A y B de todos los clientes.

Para posibilitar la compartición de las cachés entre diferentes clientes, especialmente en el caso de la caché predictiva que debe mantener las zonas de cada uno, se ha creado un mecanismo para que los clientes se registren en la caché, informándola de que debe gestionar los *buffers* para dicho cliente. Esto se realiza mediante el método `registraCliente` definido en la superclase abstracta `CacheRAM`. Cada vez que se crea un objeto `CacheTRAM`, éste se registra automáticamente en la `CacheRAM` que tiene asociada su `CargadorTRAM`.

De esta manera, `CacheRAMPrecarga` conoce la lista de clientes (`CacheTRAM`) que la utilizan y puede solicitar la posición de su centro de detalle para mantener actualizada la posición de las zonas A, B y C de cada uno.

En el apéndice G se describen algunos ejemplos de cargadores asociados a cachés de información de tipo *raster*.

F.2. Cachés de información vectorial

Las cachés en RAM de información vectorial se gestionan mediante la clase `CacheRAMSceneGraph`, que actúa en conjunto con subclases de `CargadorRAM` específicas para el manejo de información vectorial.

Ambos componentes, cache y cargador, son responsables del *render* de la información vectorial de la zona correspondiente a una tesela, y actuarán en conjunto repartiendo las tareas necesarias. Es necesario tomar la decisión acerca de las responsabilidades que se asignan a cada componente. En el diseño actual, puesto que se ha planteado variedad de modelos de *scene graph*, `CacheRAMSceneGraph` delega completamente la tarea de *render* en el cargador, de forma que cada implementación lo gestione con total libertad.

Los cargadores desarrollados en este trabajo asociados a `CacheRAMSceneGraph` incluyen `CargadorRAMSceneGraphOSG` que utiliza la librería OSG [18] para gestionar el *scene graph* y `CargadorRAMSceneGraphGisinho`, que utiliza una librería específica para el manejo de capas de información GIS vectorial 2D, denominada Gisinho y desarrollada como parte del proyecto SANTI [83]. Estos componentes están descritos en el apéndice G.

Apéndice G

Acceso a las fuentes de datos

La última pieza de la cadena descrita en el sistema de texturizado dinámico es el acceso a las fuentes de datos externas a dicho sistema.

Este módulo actúa como frontera del sistema de texturizado hacia los proveedores externos de datos. En este sentido, los cargadores actúan como clientes o controladores para el acceso a diferentes servicios o recursos, ya sean ficheros locales o peticiones a través de una red a un servidor remoto.

La interfaz de un cargador, ilustrada en el diagrama de la figura G.1, permite solicitar cualquier *buffer* a partir de sus coordenadas (fila y columna) en el espacio virtual de un nivel determinado.

El diseño de la superclase `CargadorRAM` contempla también la posibilidad de realizar postprocesos opcionales sobre los *buffers* cargados, independientemente de cuál sea su origen o el método de acceso. Esta funcionalidad se ha utilizado, entre otras cosas, para añadir información de depuración coloreando los buffers cargados mediante un código de colores en función del nivel de detalle (ver figura 5.1).

A continuación se describen los cargadores desarrollados en este trabajo, que suponen un repertorio variado con el objetivo de cubrir todas las características disponibles en el sistema de texturizado virtual dinámico así como las necesidades de uso más habituales.

G.1. CargadorRAMClipmapSGI

Este cargador obtiene los buffers directamente a partir de ficheros en disco. Cada fichero contiene un volcado directo de la memoria para una región cuadrada de textura en el formato utilizado internamente en TRAM. No se aplica ningún tipo de compresión, por lo que se dispone de la calidad original intacta aunque los requisitos de almacenamiento son elevados.

Este formato es completamente compatible con el de la implementación de *clipmaps* de SGI, disponible a partir de las arquitecturas InfiniteReality [114] e integrado con su sofware de gestión de *scene graph* OpenGL Performer

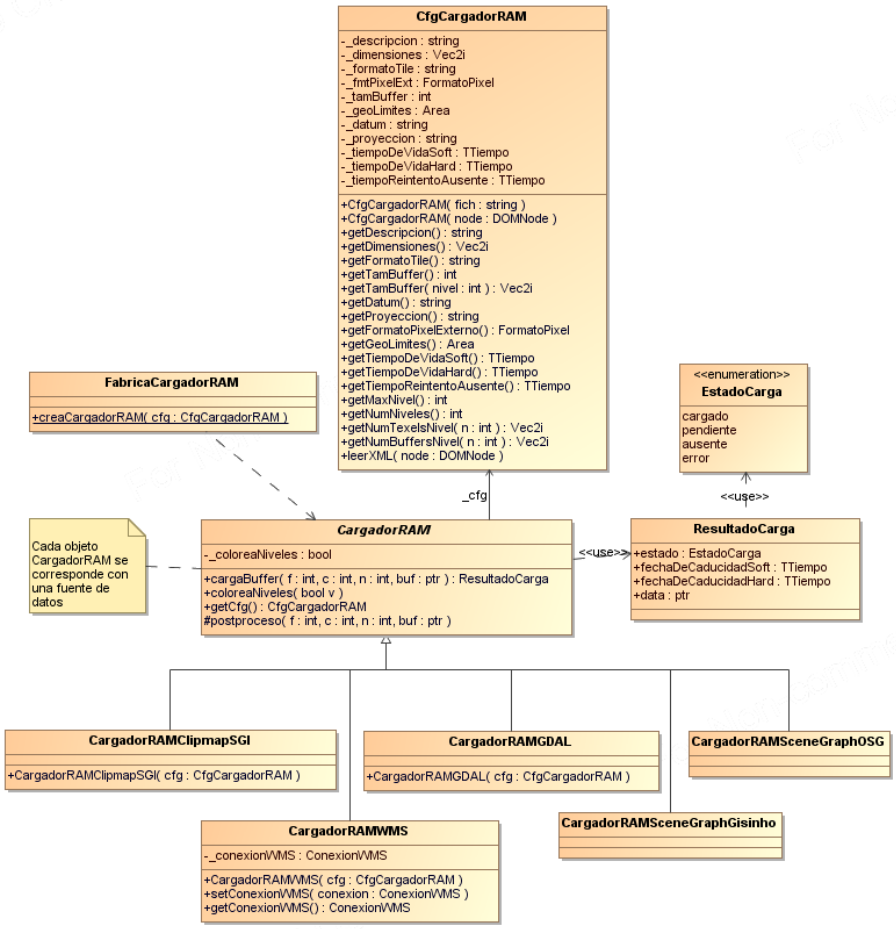


Figura G.1: Diagrama UML de la clase `CargadorRAM`.

[127], y de ahí se deriva su nombre.

Cada nivel de detalle de la pirámide virtual corresponde con un directorio. Dentro de esos directorios existe a su vez otro directorio para cada fila de buffers en que se divide el espacio virtual del nivel a efectos de la caché de segundo nivel. Dentro de esos directorios existe un fichero para el buffer correspondiente a cada columna del espacio virtual recién mencionado.

La ventaja de este cargador es que se evitan las tareas de compresión, descompresión y transformación de los datos, puesto que se almacenan en disco tal y como se disponen en memoria, pero sobre todo la mayor ventaja es que se ofrece la calidad máxima sin degradar la imagen en absoluto. No obstante, el ancho de banda necesario para las transferencias de información se dispara, lo cual es especialmente importante cuando los datos se obtienen a través de la red de un disco remoto, y sobre todo el elevado espacio de almacenamiento necesario, que puede llegar a hacer inviable esta estructura

para texturas muy extensas y de elevada resolución.

Otra característica de este cargador es que la base de datos debe prepararse de antemano, en un preproceso *off-line* que puede resultar lento debido al volumen de datos que se manejan habitualmente.

G.2. CargadorRAMGDAL

Este cargador permite el acceso a imágenes georreferenciadas, mediante el uso de la librería GDAL (Geospatial Data Abstraction Library) [8].

La gran ventaja es el acceso inmediato a cualquier imagen disponible, la contrapartida es que este acceso puede resultar lento, puesto que de la imagen fuente se debe extraer, en tiempo de ejecución, la región correspondiente al buffer escalándola al nivel de detalle adecuado y en muchos casos también se deben convertir los *texels* al formato utilizado por la textura.

En cierto modo se puede ver como un atajo a la técnica anterior (`CargadorRAMClipmapSGI`) que evita el proceso previo de generación de la base de datos y permite texturizar directamente cualquier imagen georreferenciada.

Otra ventaja importante de este cargador es que se dispone de acceso a todos los formatos de imagen soportados por GDAL, los cuales incluyen la práctica mayoría de los utilizados actualmente en el campo de los sistemas de información geográfica. Los formatos utilizados durante este trabajo han sido principalmente GeoTIFF y ECW.

En la práctica, este cargador no resulta útil en la fase de explotación debido a su pobre rendimiento, aunque sí ha resultado de gran utilidad durante el desarrollo y depuración de aplicaciones y bases de datos cartográficas.

G.3. CargadorRAMWMS

Además de los cargadores que acceden a los datos a través de un sistema de ficheros (ya sea en un disco local o remoto), se posibilita su acceso a través de Internet o de redes privadas mediante el protocolo HTTP, utilizando el servicio WMS (Web Map Service). Para ello se utiliza la clase `CargadorRAMWMS`.

En este caso, a partir de las coordenadas del buffer solicitado y el nivel de detalle, conociendo la extensión de la textura virtual, se solicitan al servidor WMS ciertas capas de información para la zona geográfica correspondiente y en la resolución determinada por el nivel de detalle que se requiere.

El servidor WMS devolverá, si no se produce ningún error, la imagen resultante en un formato gráfico determinado (PNG, JPEG, ...) y el cargador se encargará de transformar esa imagen al formato interno de la textura y almacenarlo de esta forma en la `CacheRAM`.

En estos casos la velocidad de acceso será notablemente más reducida que la de los accesos a disco de los cargadores descritos anteriormente. Además, el

servidor se verá abordado por una elevada cantidad de peticiones de imágenes realizadas por el sistema de texturizado virtual dinámico. Por estos motivos, `CargadorRAMWMS` incluye una caché local en disco que almacena los resultados de las peticiones exitosas al servidor WMS, que palía ambos problemas: mejora la velocidad y sobre todo latencia de los accesos y reduce el volumen de peticiones al servidor.

El cargador de buffers mediante WMS se apoya en unos módulos que gestionan el acceso al servidor y la configuración de las peticiones. Estos módulos se muestran en la figura G.2.

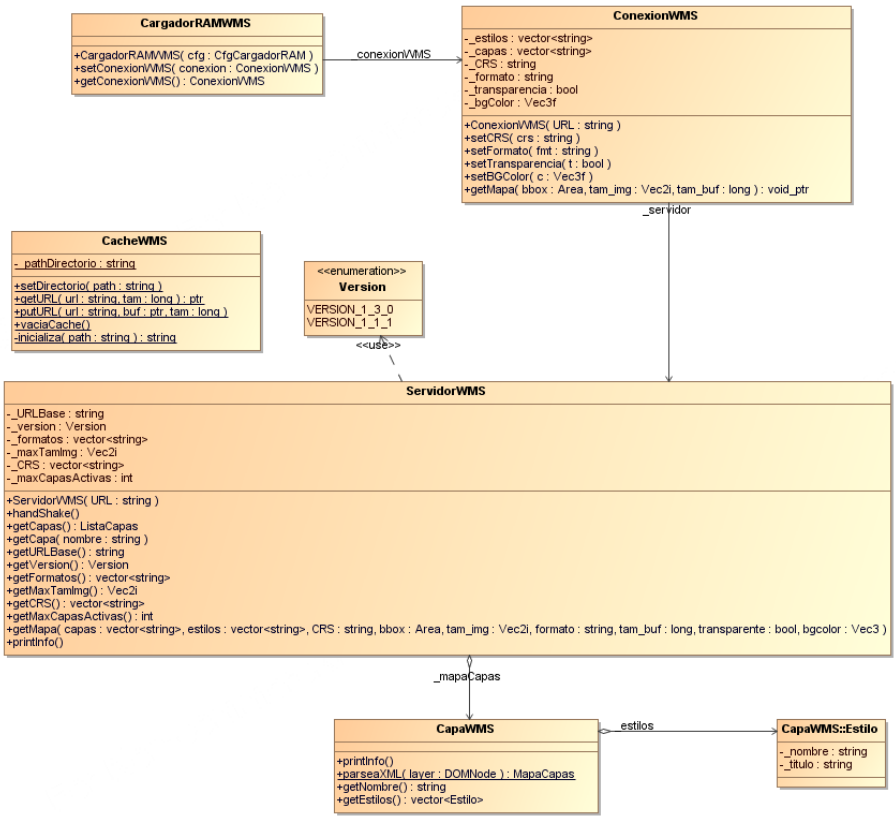


Figura G.2: Diagrama UML de la clase `CargadorRAMWMS`.

`ConexionWMS` encapsula la gestión de la conexión mediante HTTP con el servidor WMS y la configuración común a todas las peticiones que se realizarán a ese servidor (capas, estilos, formatos, transparencia, sistema de coordenadas, etc.). Se solicitarán las imágenes correspondienets a los buffers mediante el método `getMapa`. Estas solicitudes no llegarán directamente al servidor WMS, sino que pasan a través de `CacheWMS`, que actua como `proxy` [75]. Esta caché de tercer nivel proporcionará directamente la información que ya haya sido solicitada previamente al servidor y por tanto está almacenada en el disco local y solicitará al servidor aquellos *buffers* que no estén

presentes en la caché, almacenándolos para posibles futuras peticiones.

Dado que se trabaja con información dinámica, en este tercer nivel de caché se gestiona, al igual que en los dos niveles anteriores, los tiempos de vida o validez de la información almacenada, de forma que una vez expirado dicho plazo se soliciten de nuevo al servidor en caso de ser requeridos por los niveles de caché superiores. El diseño de la gestión de tiempos de caducidad de los *buffers* es equivalente al de `CacheRAM`.

G.4. CargadorRAMSceneGraphOSG

La clase `CargadorRAMSceneGraphOSG` funciona como puente de acceso rápido a cualquier escena de OSG (y por tanto a cualquier formato del cual exista un cargador en OSG). Esta conexión ofrece disponer de forma automática de todas las características de esta potente librería de gestión de *scene graphs*.

La escena de OSG será considerada únicamente en dos dimensiones, por lo que se descarta el valor de uno de los ejes. Habitualmente se considera el eje Z como la altura del terreno, por lo que ese valor es ignorado y se trabaja únicamente con vectores en 2D, correspondientes a los otros dos ejes que forman el plano XY.

Este módulo resulta útil para la visualización de una colección de datos vectoriales como un plano técnico (trazado de una autovía, plan urbanístico, etc.) con un detalle elevado. Sin embargo carece de soporte para la gestión de capas de información.

G.5. CargadorRAMSceneGraphOSGGIS

Este cargador es una extensión del anterior para añadir el soporte de capas y ciertas técnicas de optimización a la hora de visualizar de forma eficiente grandes volúmenes de datos vectoriales.

La problemática del manejo de estos datos se analiza en el apartado 4.6. Principalmente se puede resumir en los siguientes aspectos:

- Organización jerárquica de la escena para optimizar el rendimiento del *render* a textura de los datos vectoriales. Proceso de filtrado (*cull*) de información masiva.
- Diferencias en la representación de los elementos vectoriales según la escala de la visualización. Generalización cartográfica.
- Visualización dependiente de la orientación de la cámara. Rotación de textos e iconos.
- Gestión de niveles de detalle según el rendimiento del *render*. Control del límite del tiempo de actualización de la textura.

Nombre	*Raster*/Vectorial	Método acceso	Compresión
CargadorRAMClipmapSGI	R	disco/HTTP	
CargadorRAMGDAL	R	disco/HTTP	•
CargadorRAMWMS	R	HTTP(WMS)	•
CargadorRAMSceneGraphOSG	V	disco/HTTP	N/A
CargadorRAMSceneGraphOSGGIS	V	disco/HTTP	N/A
CargadorRAMSceneGraphGisinho	V	disco/HTTP	N/A

Cuadro G.1: Resumen de cargadores en RAM disponibles en el desarrollo actual.

- Gestión de capas vectoriales. Selección ágil de la información a visualizar.

G.6. CargadorRAMSceneGraphGisinho

Este módulo ha sido implementado para adaptar el motor de textura descrito en esta memoria a un sistema de manejo de capas de información geográfica ya existente. Este sistema de visualización GIS, denominado Gisinho, estaba integrado con la generación anterior del motor de texturas virtuales del proyecto SANTI[83].

Gisinho acepta ficheros en el formato shape de ESRI y se puede configurar con diferentes tipos de visualización, uno de los cuales consiste en proyectar la información vectorial 2D sobre la textura mapeada en el terreno definiendo colores, niveles de transparencia y grosores de línea por capa.

Esta implementación de CargadorRAMSceneGraph ha servido como prueba de adaptabilidad del motor desarrollado en esta tesis doctoral, verificando la facilidad de su integración en entornos ya existentes.

G.7. Resumen de los cargadores disponibles

En la tabla G.1 se resumen las características de los cargadores disponibles actualmente en el sistema de texturizado virtual dinámico, indicando si se trata de información *raster* o vectorial, local o remota, si soporta compresión y los métodos/protocolos de acceso a los datos utilizados.

Apéndice H

Gestión de la carga del sistema

H.1. La actualización dentro del ciclo de *render*

En un sistema de simulación visual en tiempo real, el bucle de *render* se sincroniza con la frecuencia de barrido de la pantalla. Esta sincronización se combina con el uso de la técnica de doble (o incluso triple) *buffer* [34]. En cada iteración del *render*, tras dibujar en el *back buffer* (no visible en pantalla), se espera a que comienze el próximo barrido o refresco de la pantalla para intercambiar (*swap*) los papeles entre ese *back buffer* y el *front buffer* visible actualmente en pantalla. Acto seguido se continuará un nuevo ciclo de *render* dibujando en el *back buffer* actual.

Por lo tanto, se produce un tiempo de espera después del *render* de cada fotograma, necesario para esa sincronización con la pantalla.

El ciclo de *render* de la aplicación interactiva, además de las órdenes de dibujado propiamente dichas, realizará una serie de tareas imprescindibles de actualización de la escena, que incluyen al menos la lectura de las órdenes del usuario a través de los dispositivos de entrada, el posicionamiento de la cámara y las modificaciones de la escena que se produzcan como respuesta a dichas órdenes.

Algunas de estas tareas se pueden paralelizar en diferentes hilos de ejecución. Los motores de *scene graph* como Performer u OSG separan, o más bien ofrecen la opción de separar en diferentes hilos, al menos las fase de actualización (*app* o *update*), la de filtrado (*cull*) y la de dibujado (*draw*). Esta última engloba todas las tareas que se deben realizar dentro del contexto gráfico.

Entre las tareas a realizar en el mismo hilo de ejecución que la fase de dibujado se incluye la actualización de las texturas virtuales que se utilicen, puesto que la actualización síncrona de la memoria de la GPU se debe realizar dentro del contexto gráfico. Típicamente estas tareas de actualización se realizan en una fase anterior a la de dibujado, de forma que el resultado final contemple todos esos cambios. El orden habitual de las tareas se ilustra en

la primera opción de la figura H.1.

Sin embargo, se produce una situación en la cual si el tiempo de actualización es excesivo, se corre el peligro de superar el tiempo del fotograma de forma que se pierde el barrido de la pantalla y hay que esperar a la siguiente actualización. Se dice en estos casos que se "pierden *frames*". Por otra parte, si el tiempo de actualización es reducido, muy probablemente en ocasiones no se alcanzará la calidad necesaria en la textura. En estos casos no se superará el tiempo del fotograma y el tiempo sobrante hasta el próximo barrido se desperdicia esperando en la fase de sincronización, representada en cian en la figura H.1. El tiempo dedicado a esta espera es un tiempo inútil que se podría aprovechar para otras tareas, especialmente cuando puede ser necesario para la actualización.

La actualización de las cachés de texturas virtuales está organizada en pequeños bloques, como se ha descrito en la sección 4.4, y en la mayor parte de los casos el tiempo de actualización de bloque será relativamente fácil de predecir, lo cual hace posible un elevado control sobre el tiempo dedicado a la actualización, que se puede adaptar al disponible. Sin embargo, el tiempo necesario para el *render* del fotograma es mucho menos predecible, y en este caso, a diferencia de la actualización, dicho tiempo no se puede acortar dejando parte para el siguiente fotograma, hay que completar siempre esa tarea en el tiempo de fotograma.

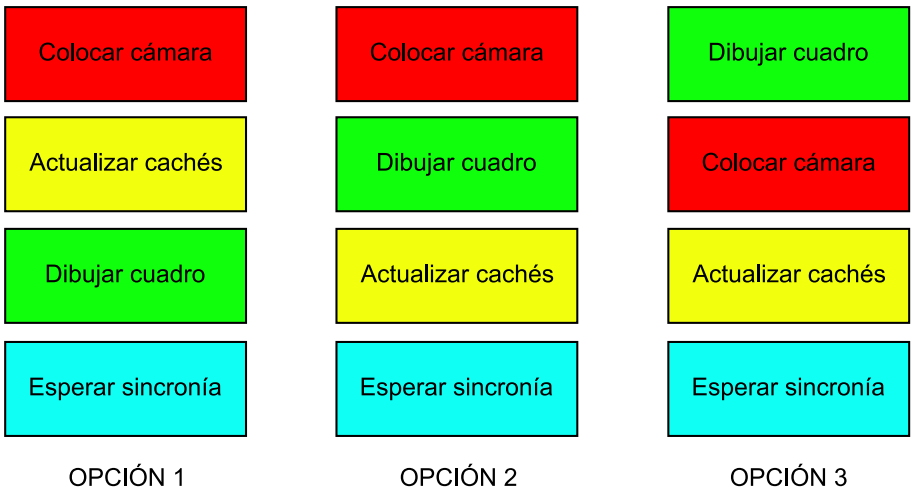


Figura H.1: Diferentes opciones para el orden de las tareas necesarias para la generación de un cuadro de la animación.

Por estos motivos, en este trabajo se ha planteado la posibilidad de realizar la actualización de las texturas virtuales dinámicas posteriormente al dibujado de la escena. Este cambio de orden proporciona la gran ventaja de que se puede conocer exactamente cuál es el tiempo disponible hasta el próximo refresco de pantalla y que por tanto se puede dedicar a la actualiza-

ción sin provocar caídas en la frecuencia de actualización y sin desperdiciar ese tiempo final en cada iteración del *render*, que de otra forma se dedicaría a esperar por el próximo barrido de pantalla.

La desventaja que se tiene con la actualización tras el dibujado es que se introduce una latencia adicional en el sistema. Esta latencia se puede producir de dos maneras, dependiendo de si el desplazamiento de la cámara (y con ella el centro de detalle) producido como respuesta a las órdenes del usuario se aplica antes del dibujado o después, junto con la actualización síncrona de las texturas virtuales.

En el primer caso (opción 2 de la figura H.1) la latencia estará en la disponibilidad del detalle de la textura. El efecto de la actualización de la textura no será visible hasta el próximo fotograma, lo cual en un sistema como el desarrollado en este trabajo no necesariamente es un problema excesivamente grave y en muchos casos no será apreciable.

En el segundo caso (opción 3 de la figura H.1) no se producirá retardo adicional en la disponibilidad del detalle de las texturas, pero sí se aumentará la latencia del posicionamiento de la cámara en respuesta a las órdenes del usuario.

H.2. Subsistema de control de carga del *render*

Con la modificación recién descrita, el tiempo que se destina a la actualización de las cachés de las texturas virtuales dinámicas dentro del ciclo de *render* se calcula en función del tiempo que tarden el resto de tareas (incluido el dibujado de la escena en el *frame buffer*) y por tanto del tiempo restante disponible hasta el próximo refresco de la pantalla.

El conocimiento de estos tiempos resulta útil no sólo para determinar el que se dedicará a la actualización sino también para anticipar posibles caídas de rendimiento en situaciones de carga elevada en el *render*.

Se ha diseñado un módulo para este control de la carga del sistema. Las tareas de este módulo se enumeran a continuación.

- Medir el tiempo empleado en las diferentes fases del *render* de cada fotograma. Se almacena tanto la duración del último fotograma como un pequeño histórico de los últimos fotogramas para poder realizar predicciones.
- Asignar el tiempo de actualización a cada una de las texturas virtuales dinámicas utilizadas en el sistema.
- Determinar situaciones de peligro de caída del rendimiento interactivo del sistema debido a una carga elevada en las tareas del ciclo de *render*.
- Reconfigurar dinámicamente el motor de texturizado para ajustar la calidad, de forma que se eviten caídas de rendimiento en situaciones

de carga elevada y al mismo tiempo se pueda mejorar la calidad en situaciones de carga baja. Uno de los parámetros más importantes que se puede reconfigurar en estos casos es el límite de muestras anisotrópicas para el filtrado de la textura, puesto que afecta notablemente al tiempo de *render*.

- Determinar la velocidad y tipo de movimiento de la cámara, para reconfigurar el comportamiento de las cachés de forma adecuada. De esta manera se puede elegir por ejemplo entre las diferentes políticas de carga predictiva (disponibilidad frente a estabilidad) que se han descrito para la caché de segundo nivel (`CacheRAM`) o la política de carga de la zona C (frentes en forma de L en la dirección de avance o anillos concéntricos en torno al centro de detalle. Ambos aspectos han sido descritos en la sección 4.4.

Los sistemas de planificación de tiempos de *render* con objeto de mantener la frecuencia de refresco de la pantalla se pueden clasificar principalmente en dos tipos: reactivos y predictivos [108]. Los reactivos ajustan la complejidad de la escena en función del tiempo de *render* del fotograma anterior, mientras que los predictivos realizan un cálculo estimativo del tiempo de *render* del próximo fotograma y en función de ese resultado ajustarán su complejidad tratando de garantizar que no se supere el tiempo del fotograma.

Esta planificación del tiempo de *render* no es exclusiva ni mucho menos del sistema de textura, sino que será responsabilidad de la aplicación y deberá tener en cuenta todos los subsistemas que la componen, incluyendo por supuesto el motor de texturas virtuales dinámicas. Dicho motor de textura está preparado para gestionar estas situaciones de carga y ajustar los tiempos de actualización, de forma que se podría integrar con cualquier planificador existente, tanto reactivo como predictivo.

Como demostración del funcionamiento del motor de texturizado en este aspecto, se ha desarrollado el planificador de ejemplo descrito anteriormente. Este planificador tiene un comportamiento predictivo que, salvo cambios muy bruscos en la trayectoria o velocidad de la cámara, garantiza que el motor de texturizado no provocará caídas en la frecuencia del *render*.

Apéndice I

Mediciones de rendimiento

En este apéndice se incluyen las tablas y gráficas detalladas de todas las mediciones realizadas para el análisis de las técnicas desarrolladas en este trabajo, tal y como se describen en el capítulo 5.

I.1. Rendimiento del *render*

I.1.1. Tiempos de *render* por test

A continuación se muestran las gráficas que ilustran las mediciones de tiempos de *render* de cada fotograma con las diferentes configuraciones *hardware* y *software* utilizadas para el análisis en las pruebas 1 a 18. Esta primera serie de pruebas se describe en la tabla 5.2. Los tiempos de *render* se representan en el eje Y, indicados en milisegundos. El eje X corresponde al número de fotograma de la ejecución de prueba. Los colores representan las diferentes configuraciones de prueba, descritas en la tabla 5.1, siendo rojo Bruinen, verde Caradhras, azul Shelob con la opción SLI desactivada en el driver y magenta esta misma con la opción SLI activada. Las dos últimas configuraciones apenas ofrecen diferencia entre ellas.

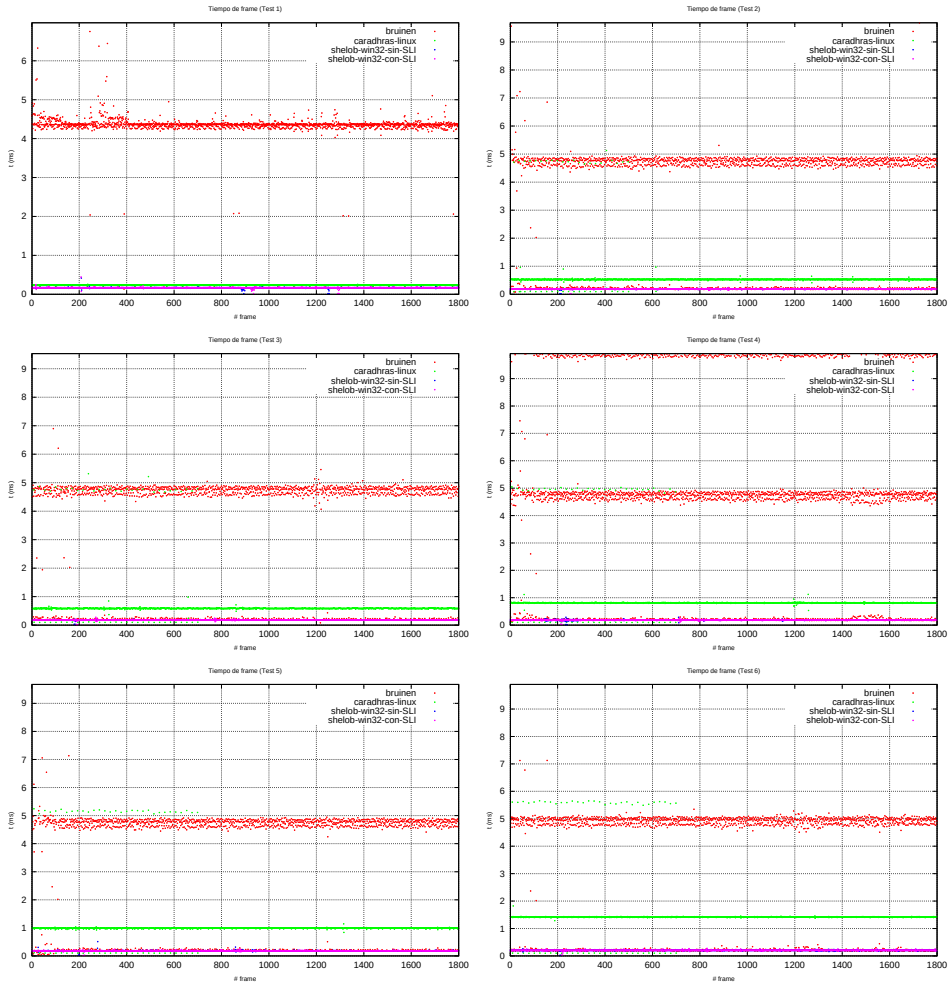


Figura I.1: Tests 1 a 6. Sin textura, OpenGL: filtrado por proximidad, bilineal, trilineal, anisotrópico 2x y anisotrópico 4x.

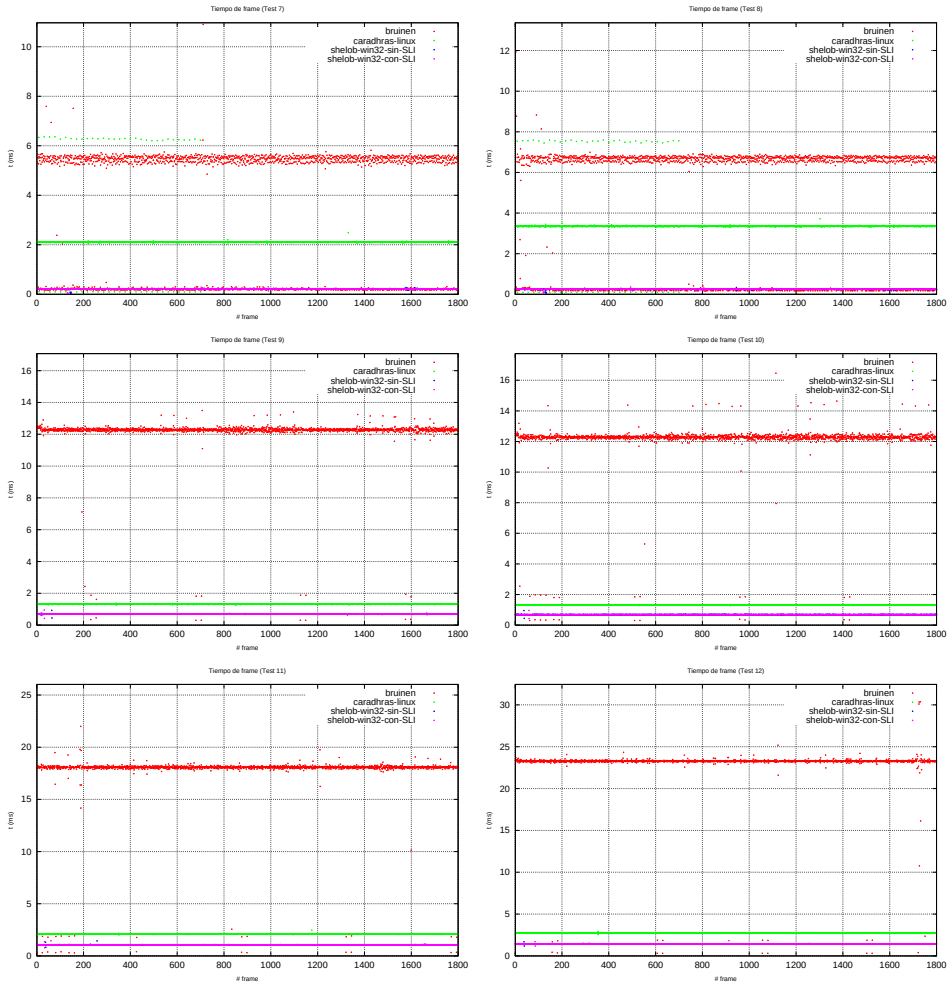


Figura I.2: Tests 7 a 12. OpenGL: filtrado anisotrópico 8x y 16x, GeoTextura: filtrado por proximidad, bilineal, trilineal y anisotrópico 1x.

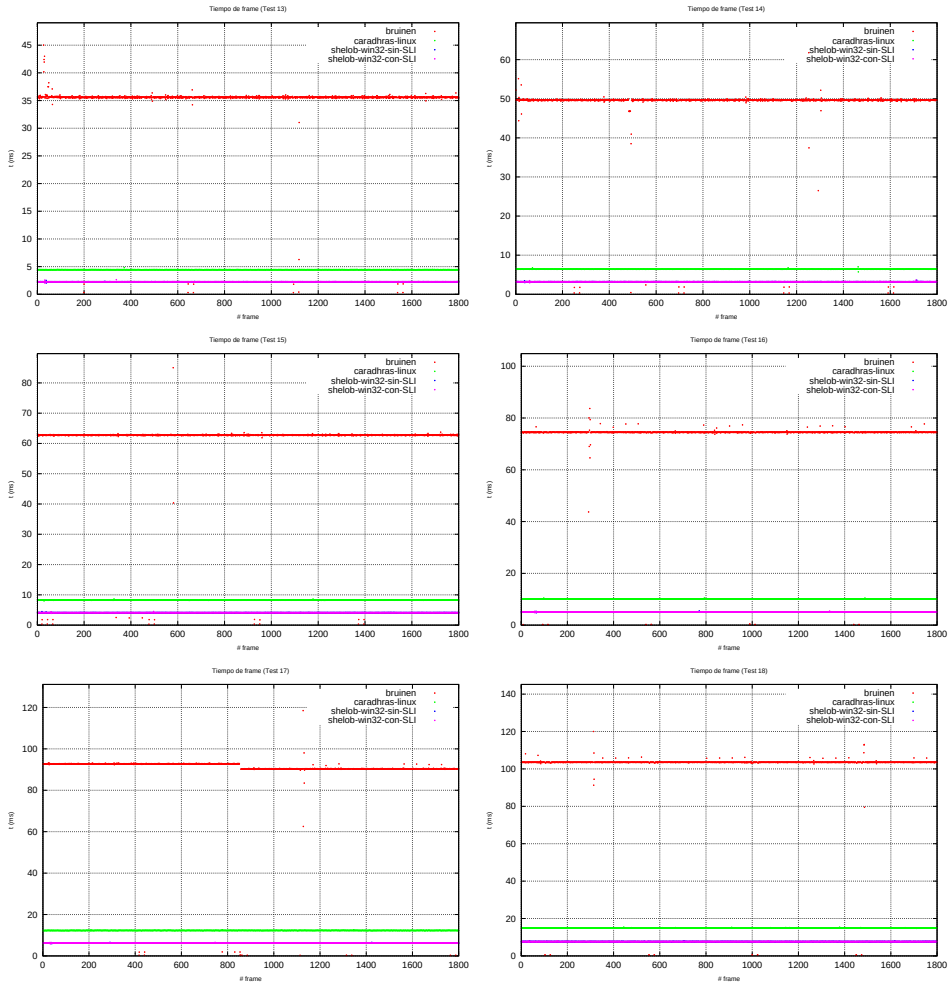


Figura I.3: Tests 13 a 18. GeoTextura: Filtrado anisotrópico 2x, 4x, 8x, 16x, 32x y 64x.

Test	t (μ)	t (σ)	fps
1	4.428	1.385	225.8
2	4.890	3.179	204.5
3	4.854	3.088	206.0
4	4.913	3.347	203.5
5	4.899	3.163	204.1
6	5.044	3.203	198.2
7	5.588	3.553	179.0
8	6.797	4.342	147.1
9	12.293	1.933	81.3
10	12.305	2.479	81.3
11	18.069	3.612	55.3
12	23.280	3.755	43.0
13	35.636	5.236	28.1
14	49.640	8.207	20.1
15	62.719	12.000	15.9
16	74.505	12.892	13.4
17	91.509	17.755	10.9
18	103.605	17.378	9.7

Cuadro I.1: Rendimiento del *render* (Bruinen).

I.1.2. Tiempos y fps por configuración

Las tablas I.1 a I.4 recopilan los valores medios de cada uno de los tests representados en las gráficas del apartado anterior. Se muestra el valor medio del tiempo de *render*, representado en milisegundos, de toda la serie de fotogramas medidos en cada prueba y para cada configuración, junto con la desviación típica de esos valores. En la última columna se incluye la frecuencia en fotogramas por segundo, calculada a partir del tiempo medio de *render*.

Test	t (μ)	t (σ)	fps
1	0.239	0.006	4189.6
2	0.573	0.455	1745.7
3	0.648	0.537	1544.2
4	0.873	0.534	1145.3
5	1.041	0.541	960.6
6	1.472	0.557	679.2
7	2.149	0.590	465.3
8	3.368	0.677	296.9
9	1.317	0.006	759.6
10	1.317	0.005	759.6
11	2.085	0.012	479.6
12	2.744	0.011	364.4
13	4.453	0.012	224.6
14	6.396	0.026	156.3
15	8.218	0.020	121.7
16	10.057	0.024	99.4
17	12.291	0.033	81.4
18	14.975	0.063	66.8

Cuadro I.2: Rendimiento del *render* (Caradhras).

Test	t (μ)	t (σ)	fps
1	0.169	0.011	5910.2
2	0.185	0.008	5394.9
3	0.185	0.007	5394.1
4	0.185	0.010	5393.3
5	0.186	0.012	5378.8
6	0.201	0.009	4983.1
7	0.216	0.008	4619.1
8	0.247	0.009	4054.0
9	0.687	0.010	1454.9
10	0.688	0.010	1454.2
11	1.070	0.014	934.8
12	1.441	0.011	694.2
13	2.280	0.009	438.6
14	3.235	0.013	309.1
15	4.131	0.014	242.1
16	5.056	0.015	197.8
17	6.174	0.027	162.0
18	7.674	0.064	130.3

Cuadro I.3: Rendimiento del *render* (Shelob).

Test	t (μ)	t (σ)	fps
1	0.170	0.012	5890.3
2	0.185	0.008	5393.3
3	0.185	0.009	5393.5
4	0.185	0.008	5393.3
5	0.186	0.009	5374.7
6	0.201	0.011	4967.3
7	0.216	0.009	4619.6
8	0.247	0.009	4041.0
9	0.687	0.010	1455.3
10	0.688	0.010	1454.5
11	1.070	0.015	934.7
12	1.441	0.015	694.0
13	2.281	0.013	438.4
14	3.235	0.013	309.1
15	4.133	0.015	241.9
16	5.058	0.016	197.7
17	6.170	0.032	162.1
18	7.675	0.062	130.3

Cuadro I.4: Rendimiento del *render* (Shelob con SLI).

Apéndice J

Comparativa de calidad de imagen

En este apéndice se incluyen las imágenes tomadas como muestra para analizar la calidad del *render* con los diferentes tipos de filtro implementados por el motor GeoTextura en comparación con las texturas de OpenGL. Las imágenes fueron tomadas con una textura de 8192×8192 *texels* y en el caso del motor GeoTextura con un tamaño de ventana de caché en TRAM de 2048×2048 *texels*.

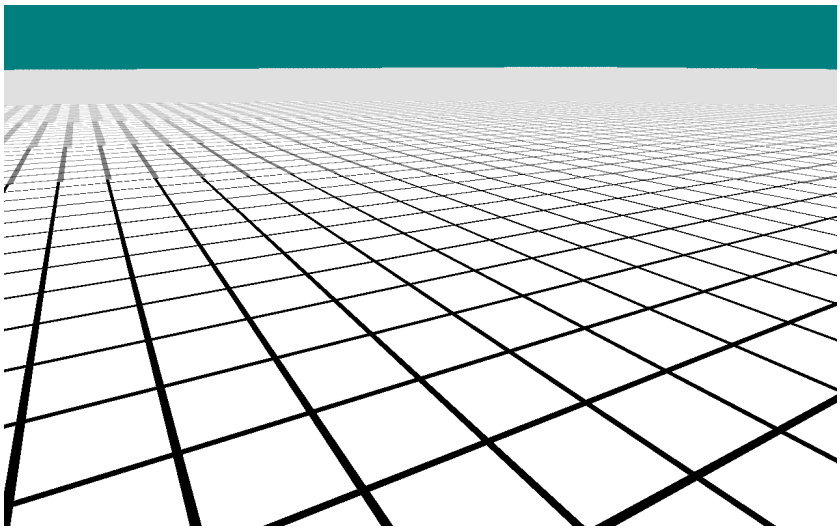


Figura J.1: GeoTextura. Filtrado por proximidad.

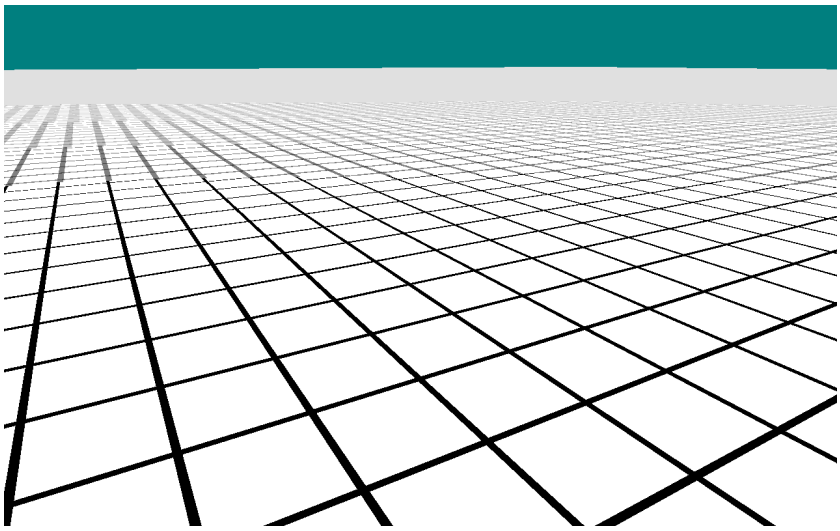


Figura J.2: OpenGL. Filtrado por proximidad.

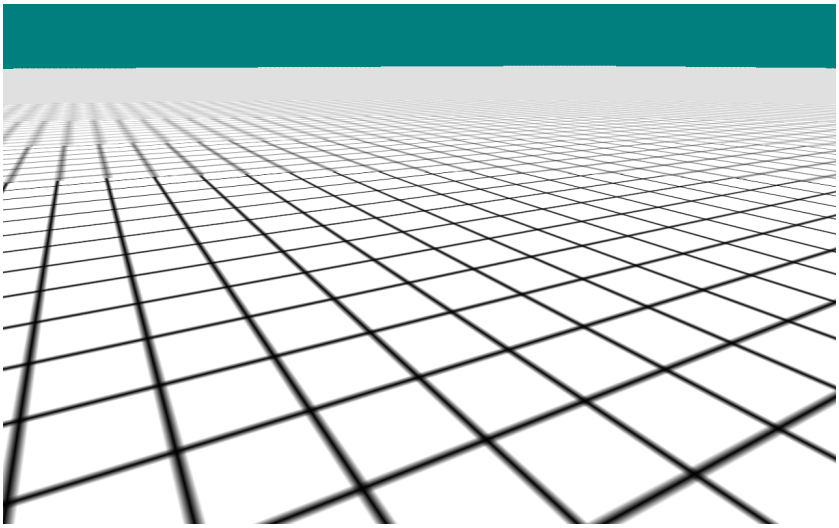


Figura J.3: GeoTextura. Filtrado bilineal.

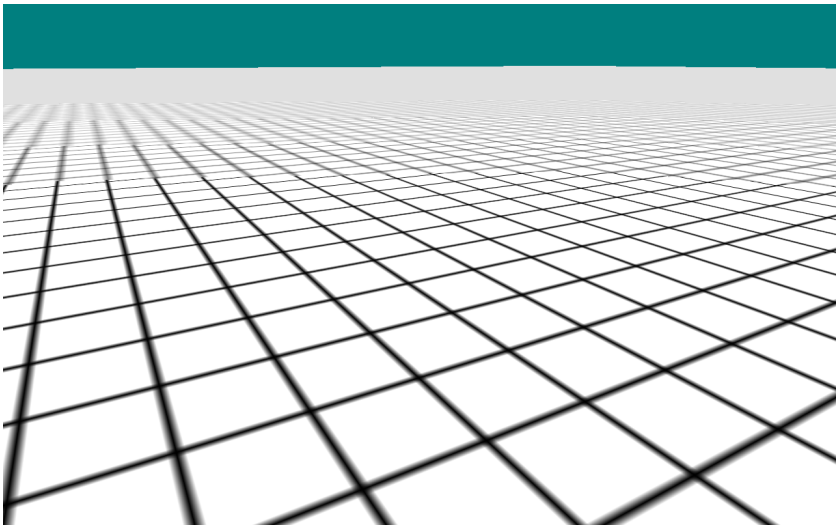


Figura J.4: OpenGL. Filtrado bilineal.

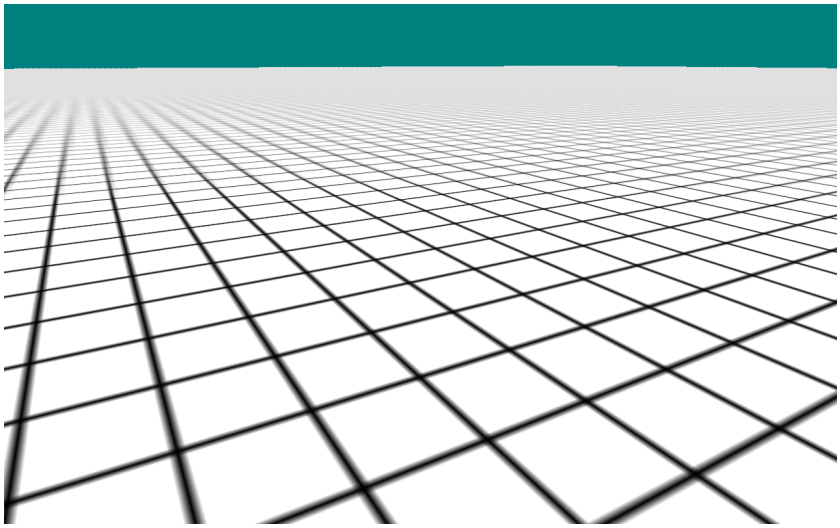


Figura J.5: GeoTextura. Filtrado trilineal.

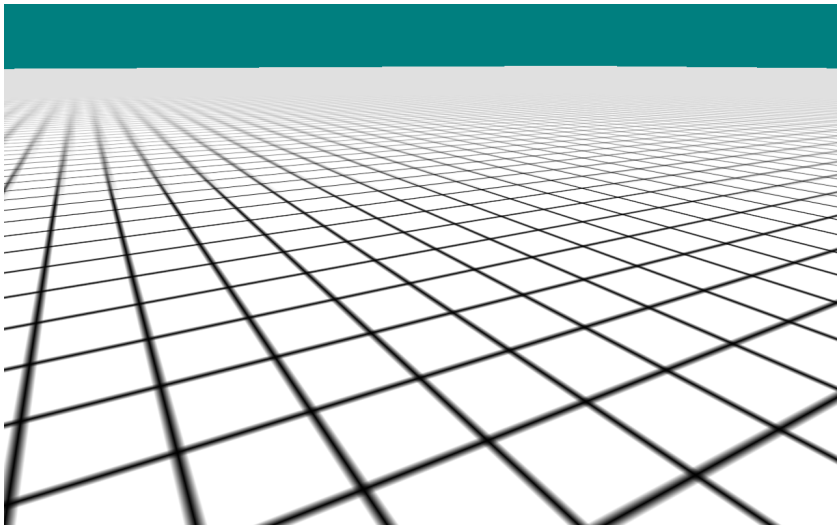


Figura J.6: OpenGL. Filtrado trilineal.

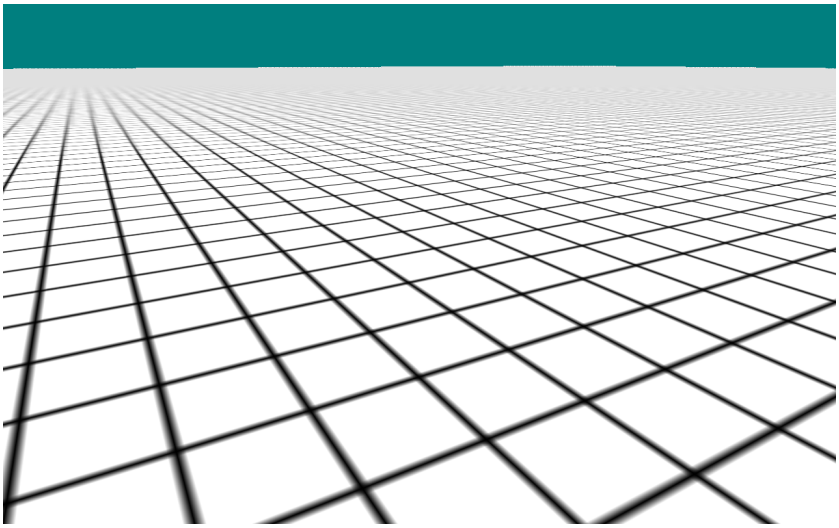


Figura J.7: GeoTextura. Filtrado anisotrópico con dos muestras.

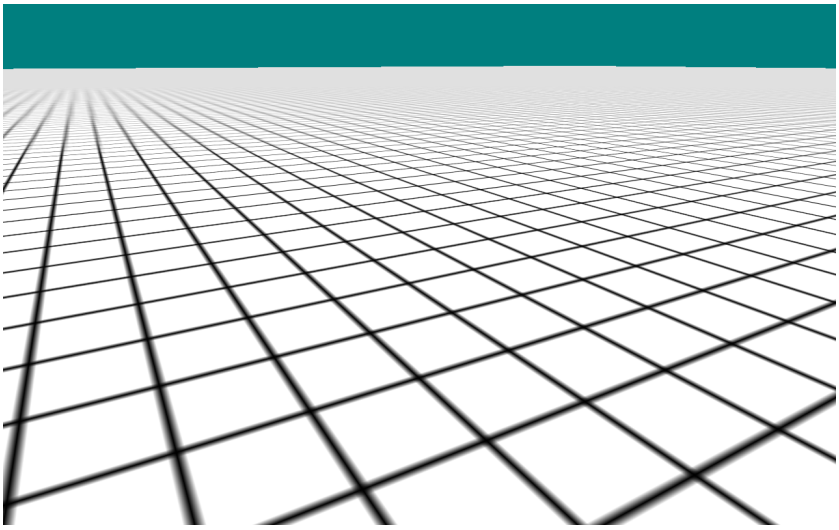


Figura J.8: OpenGL. Filtrado anisotrópico con dos muestras.

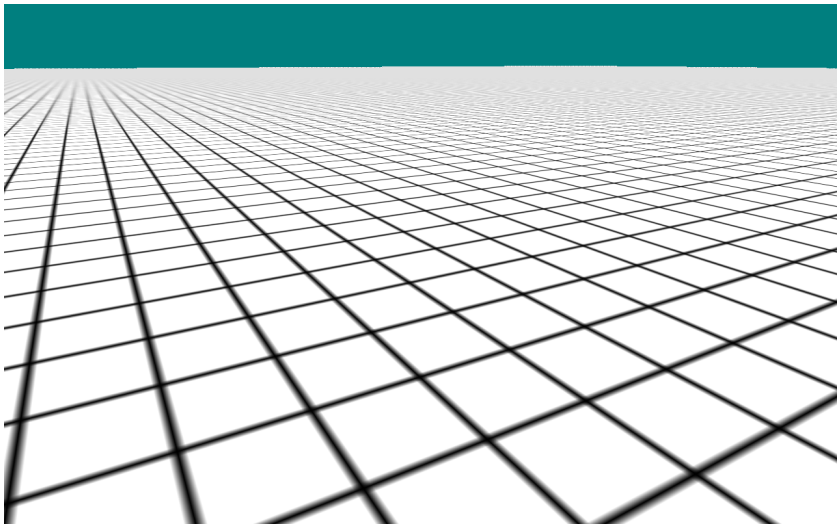


Figura J.9: GeoTextura. Filtrado anisotrópico con cuatro muestras.

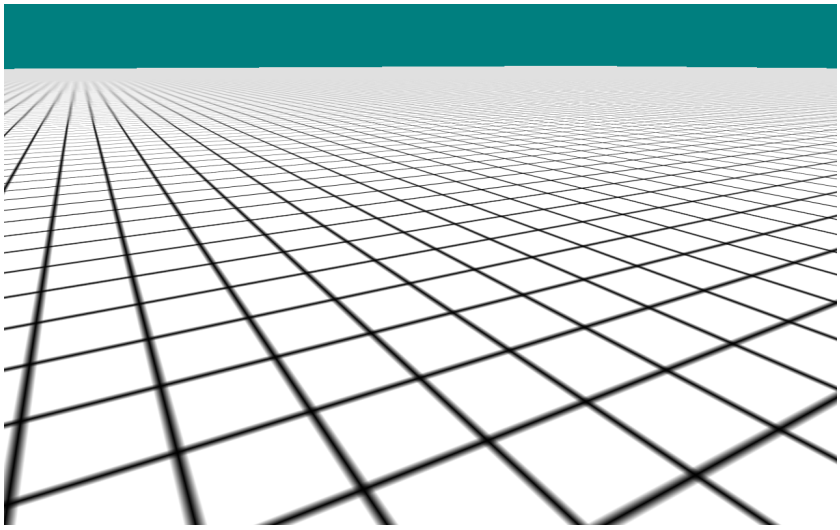


Figura J.10: OpenGL. Filtrado anisotrópico con cuatro muestras.

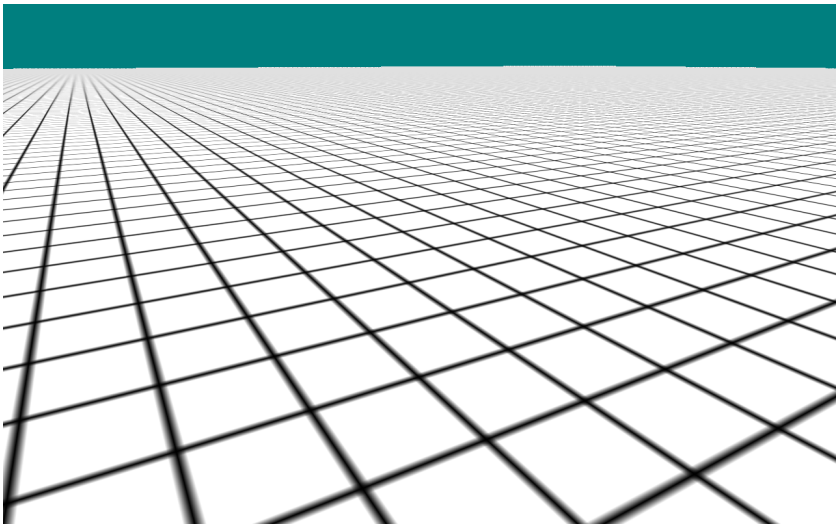


Figura J.11: GeoTextura. Filtrado anisotrópico con ocho muestras.

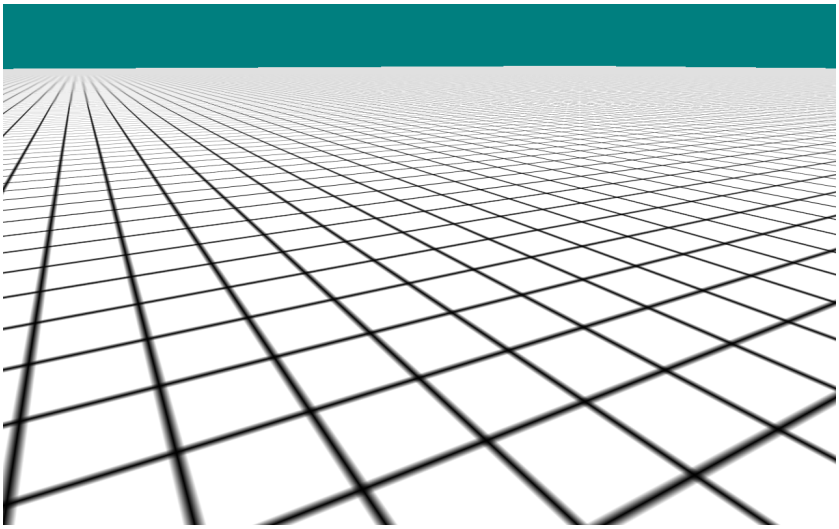


Figura J.12: OpenGL. Filtrado anisotrópico con ocho muestras.

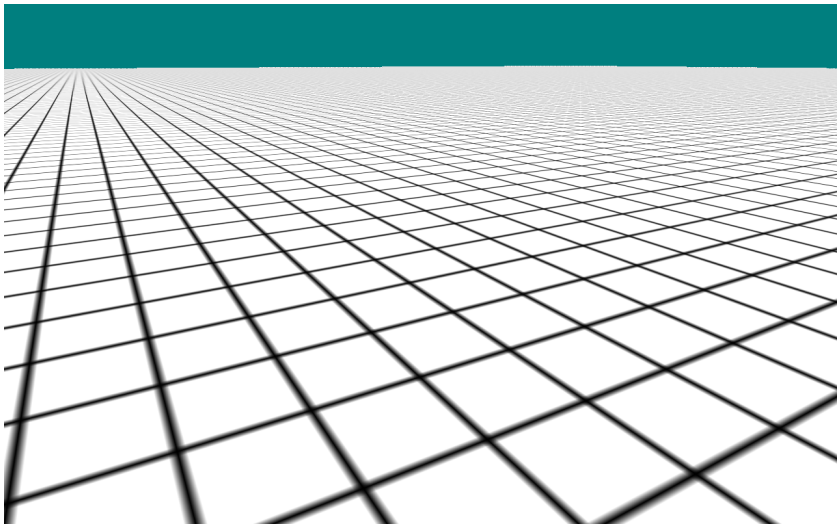


Figura J.13: GeoTextura. Filtrado anisotrópico con dieciséis muestras.

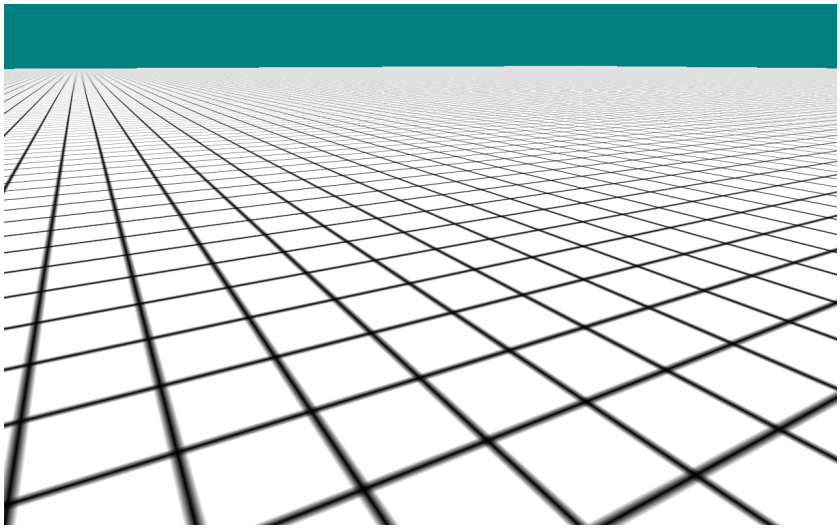


Figura J.14: OpenGL. Filtrado anisotrópico con dieciséis muestras.

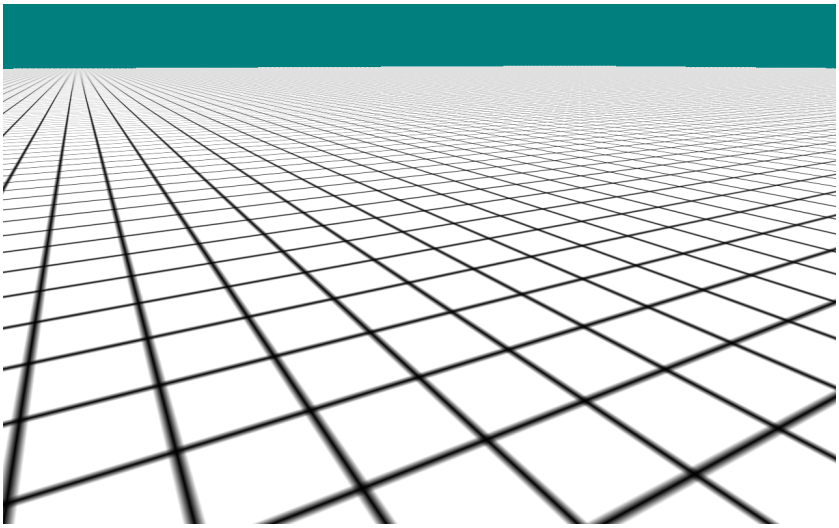


Figura J.15: GeoTextura. Filtrado anisotrópico con treinta y dos muestras.

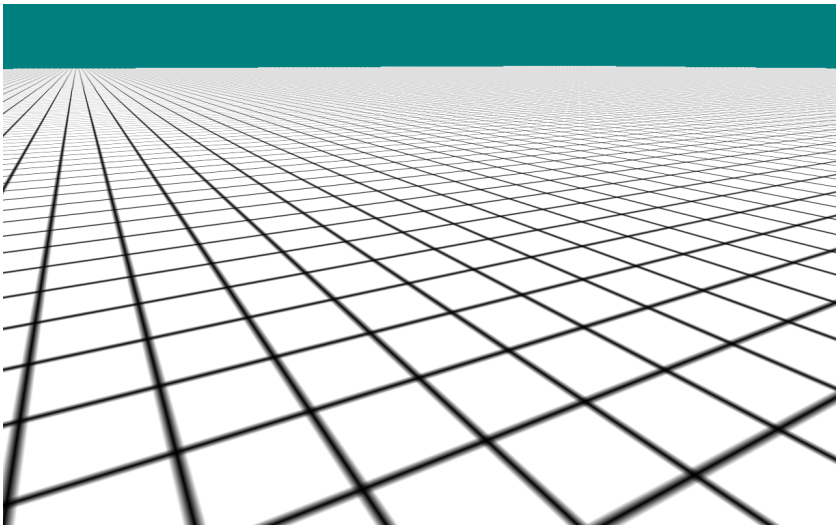


Figura J.16: GeoTextura. Filtrado anisotrópico con sesenta y cuatro muestras.

Bibliografía

[1] Autodesk. http://www.autodesk.es.

[2] Avenza Systems, Inc. http://www.avenza.com.

[3] Bentley Systems, Inc. http://www.bentley.com.

[4] Blue Marble Geographics. http://www.bluemarblegeo.com/.

[5] Clark Labs. http://www.clarklabs.org/.

[6] ER Mapper. Geospatial Imagery Solutions. http://www.ermapper.com/.

[7] ESRI - Environmental Systems Research Institute, Inc. http://www.esri.com.

[8] GDAL - Geospatial Data Abstraction Library. http://www.gdal.org/.

[9] Geographic Resources Analysis Support System. http://grass.itc.it/.

[10] GeoPISTA. http://www.geopista.com.

[11] GeoTools - The Open Source Java GIS Toolkit. http://geotools.codehaus.org/.

[12] Intergraph. http://www.intergraph.com/.

[13] Leica Geosystems. http://www.leica-geosystems.com.

[14] Leica Geosystems - Geospatial Imaging. http://gi.leica-geosystems.com.

[15] Open Geospatial Consortium (OGC). http://www.opengeospatial.org.

[16] Open Source Geospatial Foundation. http://www.osgeo.org.

[17] OpenJump. http://openjump.org/.

[18] OpenSceneGraph (OSG). http://www.openscenegraph.org.

[19] ORAD. http://www.orad.tv.

[20] OSSIM - Open Source Software Image Map. http://www.ossim.org.

[21] PostGIS. http://postgis.refractions.net/.

[22] PostgreSQL. http://www.postgresql.org.

[23] PROJ.4 - Cartographic Projections Library. http://www.remotesensing.org/proj.

[24] Quantum GIS. http://qgis.org/.

[25] Quantum3D. http://www.quantum3d.com.

[26] Skyline. http://www.skylinesoft.com.

[27] The JUMP Project. http://www.jump-project.org/.

[28] Valgrind. http://www.valgrind.org.

[29] ViewTec. http://www.viewtec.ch.

[30] Virtual Terrain Project. http://vterrain.org.

[31] María José Abásolo, Francisco Perales, and Armando De Giusti. Geometric Textured Bitree: Transmission of a Multiresolution Terrain Across the Internet. *The Journal of Computer Science and Technology (JCS&T)*, 2(7), 2002.

[32] National Aeronautics and Space Administration (NASA). World Wind. http://worldwind.arc.nasa.gov.

[33] Anupam Agrawal, M. Radhakrishna, and R. C. Joshi. Geometry-based Mapping and Rendering of Vector Data over LOD Phototextured 3D Terrain Models. In Joaquim Jorge and Vaclav Skala, editors, *The 14th International Conference in Central Europe on Computer Graphics, Visualization and Computer vision - WSCG'2006*, 2006.

[34] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering. Second Edition.* A K Peters, 2002.

[35] OpenGL ARB. ARB_texture_non_power_of_two, 2004. http://www.nvidia.com/dev_content/nvopenglspecs/GL_ARB_texture_non_power_of_two.txt.

[36] OpenGL ARB. EXT_framebuffer_multisample, 2005. http://developer.download.nvidia.com/opengl/specs/GL_EXT_framebuffer_multisample.txt.

[37] Arul Prakash Asirvatham and Hugues Hoppe. Terrain rendering using GPU-based geometry clipmaps, 2005. `http://research.microsoft.com/~hoppe/gpugcm.pdf`.

[38] Maurice J. Bach. *Design of the UNIX Operating System*. Prentice Hall, june 1986.

[39] Avi Bar-Zeev. Scenegraphs: Past, Present and Future. `http://www.realityprime.com/scenegraph.php`.

[40] Anthony C. Barkans. High quality rendering using the Talisman architecture. In *HWWS '97: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 79–88, New York, NY, USA, 1997. ACM Press.

[41] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Commun. ACM*, 19(10):542–547, 1976.

[42] David Blythe. The Direct3D 10 system. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 724–734, New York, NY, USA, 2006. ACM Press.

[43] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, august 2005.

[44] Anders Brodersen. Real-time visualization of large textured terrains. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 439–442, New York, NY, USA, 2005. ACM Press.

[45] Stephen Brooks and Jacqueline L. Whalley. A 2D/3D hybrid geographical information system. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 323–330, New York, NY, USA, 2005. ACM Press.

[46] L. M. Bugayevskiy and John Snyder. *Map Projections: A Reference Manual*. CRC, 1995.

[47] David R. Butenhof. *Programming with POSIX(R) Threads*. Addison-Wesley Professional, may 1997.

[48] Ed Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. University of Utah, 1974.

[49] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Bdam – batched dynamic adaptive meshes for high performance terrain visualization, 2003.

[50] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM). In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 20, Washington, DC, USA, 2003. IEEE Computer Society.

[51] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, 1976.

[52] David Cline and Parris K. Egbert. Interactive display of very large textures. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 343–350, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[53] Microsoft Corporation. Virtual Earth 3D. `http://maps.live.com`.

[54] NVIDIA Corporation. EXT_texture_filter_anisotropic, 1999. `http://www.nvidia.com/dev\_content/nvopenglspecs/GL\_EXT\_texture\_filter\_anisotropic.txt`.

[55] Michael A. Cosman. Global Terrain Texture: Lowering the Cost. In Eric G. Monroe, editor, *Proceedings of 1994 IMAGE VII Conference*, pages 53–64. The IMAGE Society, 1994.

[56] D. Francis Crane. Flight simulator visual-display delay compensation. In *WSC '81: Proceedings of the 13th conference on Winter simulation*, pages 59–67, Piscataway, NJ, USA, 1981. IEEE Press.

[57] Franklin C. Crow. Summed-area tables for texture mapping. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 207–212, New York, NY, USA, 1984. ACM Press.

[58] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. Surround-screen projection-based virtual reality: the design and implementation of the CAVE. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 135–142, New York, NY, USA, 1993. ACM.

[59] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti, Robert V. Kenyon, and John C. Hart. The CAVE: audio visual experience automatic virtual environment. *Commun. ACM*, 35(6):64–72, 1992.

[60] Douglass Davis, William Ribarsky, Nickolas Faust, and T. Y. Jiang. Intent, perception, and out-of-core visualization applied to terrain. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 455–458, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[61] Willem H. de Boer. Fast Terrain Rendering Using Geometrical MipMapping. 2000. `http://www.flipcode.com/articles/article\\_geomipmaps.shtml`.

[62] Sistemas Abiertos de Información Geográfica (SAIG). Kosmo. `http://www.saig.es/`.

[63] Consellería de Infraestructuras y Transporte. Generalitat Valenciana. gvSIG. `http://www.gvsig.gva.es/`.

[64] Edsger Wybe Dijkstra. Cooperating Sequential Processes, Technical Report EWD-123. Technical report, 1965.

[65] Jürgen Döllner, Konstantin Baumman, and Klaus Hinrichs. Texturing techniques for terrain visualization. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 227–234, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[66] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 81–88, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[67] Max Eckert. On the Nature of Maps and Map Logic. Translated by W. Joerg. *Bulletin of the American Geographical Society*, 40(6):344–351, 1908.

[68] Anton Ephanov and Chris Coleman. Virtual Texture: A Large Area Raster Resource for the GPU. In *The Interservice/Industry Training, Simulation and Education Conference (I/ITSEC)*. I/ITSEC, 2006.

[69] Gerald I. Evenden. Cartographic Projection Procedures for the UNIX Environment - A User's Manual. Open-File Report 90-284, United States Department of the Interior Geological Survey, Woods Hole, MA 02543, 2003.

[70] Nickolas Faust, William Ribarsky, T. Y. Jiang, and Tony Wasilewski. Real-Time Global Data Model for the Digital Earth. In *Proceedings of the International Conference on Discrete Global Grids*, 2000.

[71] Eliot A. Feibush, Marc Levoy, and Robert L. Cook. Synthetic texturing using digital filters. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 294–301, New York, NY, USA, 1980. ACM Press.

[72] György Fekete. Rendering and managing spherical data with sphere quadtrees. In *VIS '90: Proceedings of the 1st conference on Visualization '90*, pages 176–186, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[73] Leila De Floriani, Paola Magillo, and Enrico Puppo. Efficient implementation of multi-triangulations. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 43–50, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[74] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice. Second Edition in C.* Addison-Wesley, 1995.

[75] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, november 1994.

[76] GEOvirtual. GeoShow3D. `http://www.geovirtual.com`.

[77] Christopher Giertsen and Anne Lucas. 3D Visualization for 2D GIS: an Analysis of the Users' Needs and a Review of Techniques. In *Computer Graphics Forum*, volume 13, pages 1–12. Eurographics association, 1994.

[78] Brian Goldiez, Rodney Rogers, and Pam Woodard. Real-Time Visual Simulation on PCs. *IEEE Computer Graphics and Applications*, 19(1):11–15, 1999.

[79] Google. Google Earth. `http://earth.google.com`.

[80] Al Gore. The Digital Earth: Understanding our planet in the 21^{st} Century, 1998.

[81] Ned Greene and Paul S. Heckbert. Creating Raster Omnimax Images from Multiple Perspective Views Using the Eliptical Weighted Average Filter. *IEEE Computer Graphics and Applications*, 6(6):21–27, june 1986.

[82] Object Management Group. Unified Modeling Language. `http://www.uml.org`.

[83] Arquitectura y Urbanismo VideaLAB Grupo de Visualización en Ingeniería. Sistema Avanzado de Navegación sobre Terrenos Interactivo (SANTI). `http://videalab.udc.es/santi`.

[84] Michael Guthe, Aákos Balázs, and Reinhard Klein. GPU-based trimming and tessellation of NURBS and T-Spline surfaces. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1016–1023, New York, NY, USA, 2005. ACM Press.

[85] Paul Haeberli and Mark Segal. Texture mapping as a fundamental drawing primitive. In *Fourth Eurographics Workshop on Rendering*, pages 259–266. Eurographics, June 1993.

[86] Paul S. Heckbert. Survey of Texture Mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, 1986.

[87] Paul S. Heckbert. *Fundamentals of Texture Mapping and Image Warping*. University of California, Berkeley, 1989.

[88] Luis Hernández, Javier Taibo, and Antonio Seoane. Una aplicación para la navegación en tiempo real sobre grandes modelos topográficos. In *Actas del IX Congreso Español de Informática Gráfica (CEIG'99)*, pages 287–302. Universidad de Jaén, junio 1999.

[89] Alex Holkner. Hardware Based Terrain Clipmapping, 2004. `http://yallara.cs.rmit.edu.au/~aholkner/rr/ah-terrain.pdf`.

[90] Hugues Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 189–198, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[91] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 35–42, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[92] Tobias Hüttner. High Resolution Textures. In *Visualization '98 - Late Breaking Hot Topics Papers*, pages 13–17, November 1998.

[93] Tobias Hüttner and Wolfgang Strasser. Fast footprint MIPmapping. In *HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 35–44, New York, NY, USA, 1999. ACM Press.

[94] Tobias Hüttner and Wolfgang Straßer. FlyAway: a 3D terrain visualization system using multiresolution principles. *Computers & Graphics*, 23(4):479–485, 1999.

[95] Edward Imhof. *Cartographic Relief Presentation*. Walter de Gruyter & Co, 1982.

[96] Oliver Kersting and Jürgen Döllner. Interactive 3D visualization of vector data in GIS. In *GIS '02: Proceedings of the 10th ACM international symposium on Advances in geographic information systems*, pages 107–112, New York, NY, USA, 2002. ACM Press.

[97] John Kessenich. The OpenGL Shading Language (version 1.20). `http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf`.

[98] Reinhard Klein and Andreas Schilling. Efficient Multiresolution Models for Progressive Terrain Rendering. *it - Information Technology*, 44(6):314–321, 2002.

[99] David Koller, Peter Lindstrom, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory Turner. Virtual GIS: A Real-Time 3D Geographic Information System. In *VIS '95: Proceedings of the 6th conference on Visualization '95*, page 94, Washington, DC, USA, 1995. IEEE Computer Society.

[100] Wolfgang Kresse and Kian Fadaie. *ISO Standards for Geographic Information*. Springer, 2004.

[101] G. Li. Automatic Cartographic Generalization in the Digital Time. In *Proceedings of the 21st International Cartographic Conference (ICC)*. International Cartographic Association (ICA), 2003.

[102] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, and N. Faust. An integrated global gis and visual simulation system, 1998.

[103] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 109–118, New York, NY, USA, 1996. ACM Press.

[104] Peter Lindstrom and Valerio Pascucci. Visualization of large terrains made easy. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 363–371, Washington, DC, USA, 2001. IEEE Computer Society.

[105] Curt Lisle and Michelle Sartor. A Characterization of Low Cost Simulator Image Generation Systems (IST-CR-98-02). Technical report, september 1997.

[106] Paul A. Longley, Michael F. Goodchild, David J. Maguire, and David W. Rhind. *Geographic Information Systems and Science. Second Edition*. John Wiley & Sons, Ltd., 2005.

[107] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 769–776, New York, NY, USA, 2004. ACM Press.

[108] David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers, 2003.

[109] Joel McCormack, Ronald Perry, Keith I. Farkas, and Norman P. Jouppi. Feline: fast elliptical lines for anisotropic texture mapping. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 243–250, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[110] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Professional, may 1996.

[111] Robert B. McMaster and K. Stuart Shea. *Generalization in Digital Cartography*. Association of American Geographers, 1992.

[112] J. Mellaart. Excavations of Catal Hyük 1963, Anatolian Studies. *Journal of the British Institute at Ankara*, XIX, 1964.

[113] MetaVR. MetaVR 3D Layering Control Plugin for ArcMap. `http://www.metavr.com/products/arcmap\_layercontrol\_plugin.html`.

[114] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. InfiniteReality: a real-time graphics system. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 293–302, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[115] Instituto Geográfico Nacional. Ha comenzado la ejecución del Plan Nacional de Ortofotografía Aérea (PNOA), julio-septiembre 2004.

[116] NASA Earth Observatory. Blue Marble next generation. `http://earthobservatory.nasa.gov/Features/BlueMarble/`.

[117] Alan Norton, Alyn P. Rockwood, and Philip T. Skolmoski. Clamping: A method of antialiasing textured surfaces by bandwidth limiting in object space. In *SIGGRAPH '82: Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, pages 1–8, New York, NY, USA, 1982. ACM Press.

[118] NVIDIA. Clipmaps - White Paper (WP-03017-001_v01), february 2007. `http://developer.download.nvidia.com/SDK/10/direct3d/Source/Clipmaps/doc/Clipmaps.pdf`.

[119] Harry Nyquist. Certain topics in telegraph transmission theory. In *Trans. AIEE*, volume 47, pages 617–644, 1928.

[120] Marc Olano, Shrijeet Mukherjee, and Angus Dorbie. Vertex-based anisotropic texturing. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 95–98, New York, NY, USA, 2001. ACM Press.

[121] NVIDIA Corporation Pat Brown. EXT_texture_array, 2008. `http://developer.download.nvidia.com/opengl/specs/GL\_EXT\_texture\_array.txt`.

[122] D. Perny, M. Gangnet, and Ph. Coueignoux. Perspective mapping of planar textures. *SIGGRAPH Comput. Graph.*, 16(1):70–100, 1982.

[123] Boris Rabinovich and Craig Gotsman. Visualization of large terrains in resource-limited computing environments. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 95–102, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[124] Theresa Marie Rhyne, William Ivey, Loey Knapp, Peter Kochevar, and Tom Mace. Visualization and geographic information system integration: what are the needs and the requirements, if any? In *IEEE Conference on Visualization, 1994*, pages 400–403. IEEE Computer Society Press, 1994.

[125] Andrew Robinson, Katerina Mania, and Philippe Perey. Flight simulation: research challenges and user assessments of fidelity. In *VRCAI '04: Proceedings of the 2004 ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry*, pages 261–268, New York, NY, USA, 2004. ACM Press.

[126] Arthur H. Robinson. *Elements of Cartography, 2nd Edition*. John Wiley and Sons, Inc., 1960.

[127] John Rohlf and James Helman. IRIS performer: a high performance multiprocessing toolkit for real-time 3D graphics. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 381–394, New York, NY, USA, 1994. ACM Press.

[128] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition) (OpenGL)*. Addison-Wesley Professional, february 2006.

[129] Stefan Röttger, Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. Real-Time Generation of Continuous Levels of Detail for Height Fields. In *Proceedings of 1998 International Conference in Central Europe on Computer Graphics and Visualization*, pages 315–322, 1998.

[130] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[131] Andreas Schilling, G. Knittel, and Wolfgang Strasser. Texram: a smart memory for texturing. *IEEE Computer Graphics and Applications*, 16(3):32–41, may 1996.

[132] Arne Schilling, Jens Basanow, and Alexander Zipf. Vector Based Mapping of Polygons on Irregular Terrain Meshes for Web 3D Map Services. In *3rd International Conference on Web Information Systems and Technologies (WEBIST)*, march 2007.

[133] M. Schneider, M. Guthe, and R. Klein. Real-time Rendering of Complex Vector Data on 3D Terrain Models. In H. Thwaites, editor, *The 11th International Conference on Virtual Systems and Multimedia (VSMM2005)*, pages 573–582. ARCHAEOLINGUA, October 2005.

[134] Martin Schneider and Reinhard Klein. Efficient and Accurate Rendering of Vector Data on Virtual Landscapes. In *The 15-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2007*, 2007.

[135] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Versión 2.1). Technical report, 2006. `http://www.opengl.org/registry/doc/glspec21.20061201.pdf`.

[136] Antonio Seoane. Desarrollo de sistemas de visualización en tiempo real de grandes superficies de terreno sobre entornos computacionales de bajo coste. Proyecto fin de carrera de Ingeniería Informática, 2004.

[137] Antonio Seoane, Javier Taibo, Luis Hernández, Rubén López, and Alberto Jaspe. Hardware Independent Clipmapping. In *WSCG '2007: The 15^{t}h International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2007 - Full Papers Proceedings II*, pages 177–183. Eurographics Association, 2007.

[138] Claude E. Shannon. Communication in the presence of noise. In *Proc. Insitute of Radio Engineers*, volume 37, pages 10–21, 1949.

[139] Hyun-Chul Shin, Jin-Aeon Lee, and Lee-Sup Kim. SPAF: sub-texel precision anisotropic filtering. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 99–108, New York, NY, USA, 2001. ACM Press.

[140] James R. Smith. *Introduction to Geodesy. The History and Concepts of Modern Geodesy.* Wiley Series in Surveying and Boundary Control, Roy Minnick, Series Editor. John Wiley & Sons, Inc., 1997.

[141] John P. Snyder. *Map Projections: A Working Manual.* US Geological Survey, US Government Printing Office, 1987.

[142] John P. Snyder and Philip M. Voxland. *Album of Map Projections.* Diane Pub., 1986.

[143] Rebecca R. Springmeyer, Meera M. Blattner, and Nelson L. Max. A Characterization of the Scientific Data Analysis Process. In *IEEE Conference on Visualization, 1992. Visualization '92, Proceedings*, pages 235–242. IEEE Computer Society Press, october 1992.

[144] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: a virtual mipmap. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 151–158, New York, NY, USA, 1998. ACM Press.

[145] Roland Wahl, Manuel Massing, Patrick Degener, Michael Guthe, and Reinhard Klein. Scalable Compression and Rendering of Textured Terrain Data. In *WSCG*, pages 521–528, 2004.

[146] Zachary Wartell, Eunjung Kang, Tony Wasilewski, William Ribarsky, and Nickolas Faust. Rendering vector data over global, multi-resolution 3D terrain. In *VISSYM '03: Proceedings of the symposium on Data visualisation 2003*, pages 213–222, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[147] Jr. William Dungan, Anthony Stenger, and George Sutty. Texture tile considerations for raster graphics. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 130–134, New York, NY, USA, 1978. ACM Press.

[148] Lance Williams. Pyramidal parametrics. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 1–11, New York, NY, USA, 1983. ACM Press.

[149] Tian yue Jiang, William Ribarsky, Tony Wasilewski, Nickolas Faust, Brendan Hannigan, and Mitchel Parry. Acquisition and Display of Real-Time Atmospheric Data on Terrain. In *Joint Eurographics - IEEE TCVG Symposium on Visualization*, pages 15–24, 2001.